# NAVAL POSTGRADUATE SCHOOL

2

## Monterey, California

# THESIS

DATABASE CREATION AND/OR REORGANIZATION
OVER MULTIPLE DATABASE BACKENDS

by

Deborah A. McGhee

June 1989

Thesis Advisor: David K. Hsiao

Approved for public release; distribution unlimited

REPORT DOCUMENTATION PAGE

1a Report Security Classification UNCLASSIFIED

1b Restrictive Markings

2a Security Classification Authority

2b Declassification/Downgrading Schedule

3 Distribution Availability of Report
Approved for public release; distribution is unlimited.

4 Performing Organization Report Number(s)

5 Monitoring Organization Report Number(s)

6a Name of Performing Organization
Naval Postgraduate School

6b Office Symbol *(If Applicable)* 52

7a Name of Monitoring Organization
Naval Postgraduate School

6c Address *(city, state, and ZIP code)*
Monterey, CA 93943-5000

7b Address *(city, state, and ZIP code)*
Monterey, CA 93943-5000

8a Name of Funding/Sponsoring Organization

8b Office Symbol *(If Applicable)*

9 Procurement Instrument Identification Number

8c Address *(city, state, and ZIP code)*

10 Source of Funding Numbers

| Program Element Number | Project No | Task No | Work Unit Accession No |
|---|---|---|---|
| | | | |

11 Title *(Include Security Classification)*
DATABASE CREATION AND/OR REORGANIZATION OVER MULTIPLE DATABASE BACKENDS

12 Personal Author(s)
McGhee, Deborah A.

13a Type of Report
Master's Thesis

13b Time Covered
From To

14 Date of Report *(year, month,day)*
June 1989

15 Page Count
103

16 Supplementary Notation The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17 Cosati Codes

| Field | Group | Subgroup |
|---|---|---|
| | | |
| | | |

18 Subject Terms *(continue on reverse if necessary and identify by block number)*
Multi-Lingual Database System (MLDS), Multi-Model Database System (MMDS), Multi-Backend Database System (MBDS)

19 Abstract *(continue on reverse if necessary and identify by block number*

To create a record in a database, one uses the INSERT command. However, in the Multi-Backend Database System (MBDS), the insert command only inserts one record at a time. When creating very large databases consisting of thousands or millions of records, the use of the INSERT command is a time-consuming process.

Once a database is created, some of the records of the database may be tagged for deletion. MBDS uses the DELETE command to tag these records. Over some period of time, those records tagged for deletion should be physically removed from the database. Hence, removing tagged records is in essence creating new databases from untagged records.

20 Distribution/Availability of Abstract
☒ unclassified/unlimited □ same as report □ DTIC users

21 Abstract Security Classification
UNCLASSIFIED

22a Name of Responsible Individual
Prof. David K. Hsaio

22b Telephone *(Include Area code)*
(408) 646-2253

22c Office Symbol
Code 52Hq

83 APR edition may be used until exhausted
All other editions are obsolete

19 Continued:

In this thesis, we present a methodology to efficiently create very large databases in gigabytes on parallel computers and to reorganize them when they have been tagged for deletion. Specifically, we design a utility program to by-pass the system's INSERT command, to load the data of the database directly on to disks and create all the necessary base data and meta data of the database.

# Database Creation and/or Reorganization over Multiple Database Backends

by

Deborah A. McGhee
Lieutenant, United States Navy
B.S., University of South Carolina, 1982

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1989

Author: Deborah A. McGhee
Deborah A. McGhee

Approved by:
David K. Hsiao, Thesis Advisor

Thomas C. Wu, Second Reader

Robert B. McGhee
Robert B. McGhee, Chairman
Department of Computer Science

K.T. Marshall
Kneale T. Marshall
Dean of Information and Policy Sciences

# ABSTRACT

To create a record in a database, one uses the INSERT command. However, in the Multi-Backend Database System (MBDS), the insert command only inserts one record at a time. When creating very large databases consisting of thousands or millions of records, the use of the INSERT command is a time-consuming process.

Once a database is created, some of the records of the database may be tagged for deletion. MBDS uses the DELETE command to tag these records. Over some period of time, those records tagged for deletion should be physically removed from the database. Hence, removing tagged records is in essence creating new databases from untagged records.

In this thesis, we present a methodology to efficiently create very large databases in gigabytes on parallel computers and to reorganize them when they have been tagged for deletion. Specifically, we design a utility program to by-pass the system's INSERT command, to load the data of the database directly onto disks and create all the necessary base data and meta data of the database.

# TABLE OF CONTENTS

# LIST OF FIGURES

# I. INTRODUCTION

## A. MOTIVATION

In order to create a record in databases, an INSERT command is used. In these cases, inserting records into any database requires input/output (I/O) operations. The record is read from one secondary storage, processed by the database system, then stored onto another secondary storage. We notice that there are three operations per record. The insert command is usually efficient for single record insertion, as well as a small batch of insert operations. The problems arise when massive amounts of records are to be loaded, generating thousands upon thousands of I/O requests. When this occurs, the loading of a new database is a time-consuming process. Hence, the building of a utility package that by-passes the system's INSERT command and loads the data directly onto the secondary storage would be a viable solution to help creating databases more quickly.

Over the course of time, some records in the database are no longer required and are deleted from the database. Most operational systems do not physically remove records at that point in time when these records have been identified for deletion. Instead, systems tag them so that they may be deleted later. Most systems employ a "garbage collection" routine. Although, garbage collection process imposes additional overhead and complexity on the system [Ref. 1:p. 348], it does not remove tagged records at the time of deletion. Instead, the storage reclamation [Ref. 2:p 432] is done at non-prime times. It identifies all the unused/unnecessary cells (i.e., storage space occupied by tagged records) and returns them to free storage. The garbage collection allows that a record is only tagged and its space is not reclaimed at every instance when the record is identified for deletion The recognition and disposal [Ref. 3:p. 396] of tagged records within any database is essential to the optimization of the records on the secondary storage. Good optimization produces good data organization, which, in turn, provides efficient data retrieval. Records that are tagged for deletion are removed from the database and the active records are reorganized on the secondary storage to fill the space vacated by the removed records. Removing tagged records is in essence creating a new database from the untagged records.

This thesis will provide a detailed design specification for a utility package capable of creating and/or reorganizing large databases over multiple disks. The development of this utility package is in direct support of the multi-backend database system (MBDS), which will be discussed in more detail in the next section. This utility package will provide a more efficient means for generating new databases as well as provide a garbage-collection capability.

## B. BACKGROUND

Over the past two decades, database design and implementation methods were fairly standard. The general approach was to specify a data model, define a data language for that particular model, and develop a system to manage and execute transactions written in the data language. This approach lead to the development of homogeneous database systems, which restricts the user to a single data model with its corresponding data language. Some examples of systems using the homogeneous database system approach are IBM's Information Management System (IMS) which supports the hierarchical data model and the Data Language I (DL/I), Sperry Univac's DMS-1100 supporting the network data model and the CODASYL data language, IBM's SQL/Data System supporting the relational data model and the Structured English Query Language (SQL), and Computer Corporation of America's Local Data Manager (LDM), which supports the functional data model and the Daplex data language.

A revolutionary approach to database management system development is the multi-lingual database system (MLDS), which eliminates the restrictions discussed previously [Ref. 4]. The design of MLDS affords the user the ability to access and manipulate several different databases, using their corresponding data models with their respective data languages. The major design goal of MLDS is the development of a system which can be accessed via different data models and their model-based data languages (e.g., hierarchical/DL/I, relational/SQL, network/CODASYL, and functional/Daplex). MLDS will function as a heterogeneous collection of databases vice a single database system.

The many advantages of MLDS are its ability to support a wide variety of databases using different data models and languages, economy and efficiency of hardware upgrades, and the reusability of database transactions developed on a conventional system.

MLDS has taken further steps toward a more complete utilization of its resident databases. Currently, all data models are allowed access to the database only through their corresponding languages: hierarchical databases are only accessible through DL/I, network databases are only accessible through CODASYL, functional databases are only accessible through Daplex, and relational databases are only accessible through SQL. MLDS extends the concept of a multi-lingual database systems to a multi-model database system (MMDS) in which the databases based on different models can be accessed by data languages based on different data models. This type of environment allows the user of one data model to access and manipulate data stored in another data model. The obvious benefit of MMDB is the cross-access of databases based on different models which allows true sharing of data over multiple databases.

The following subsections will give the reader an overview of the structure and function of MLDS. We also introduce the reader to the architecture of the multi-backend database system (MBDS). MBDS is the database system used by MLDS to support database transaction processing.

### 1. The Multi-Lingual Database System

A block diagram of the multi-lingual database system (MBDS) is shown in Figure 1.1. To access or modify the database, the user issues transactions through the language interface layer (LIL) using a user data model (UDM) written in a user data language (UDL) for that particular model. LIL routes the transaction to the kernel mapping system (KMS). KMS performs one of two tasks, depending on the type of database transaction requested.

If the transaction specified by the user is for the creation of a new database, KMS transforms the UDM database definition to an equivalent kernel data model (KDM) database definition. KDM database definition is sent to the kernel controller (KC), which then routes the request to the kernel database system (KDS) for processing. Upon completion, KDS notifies KC, which in turn forwards its results to

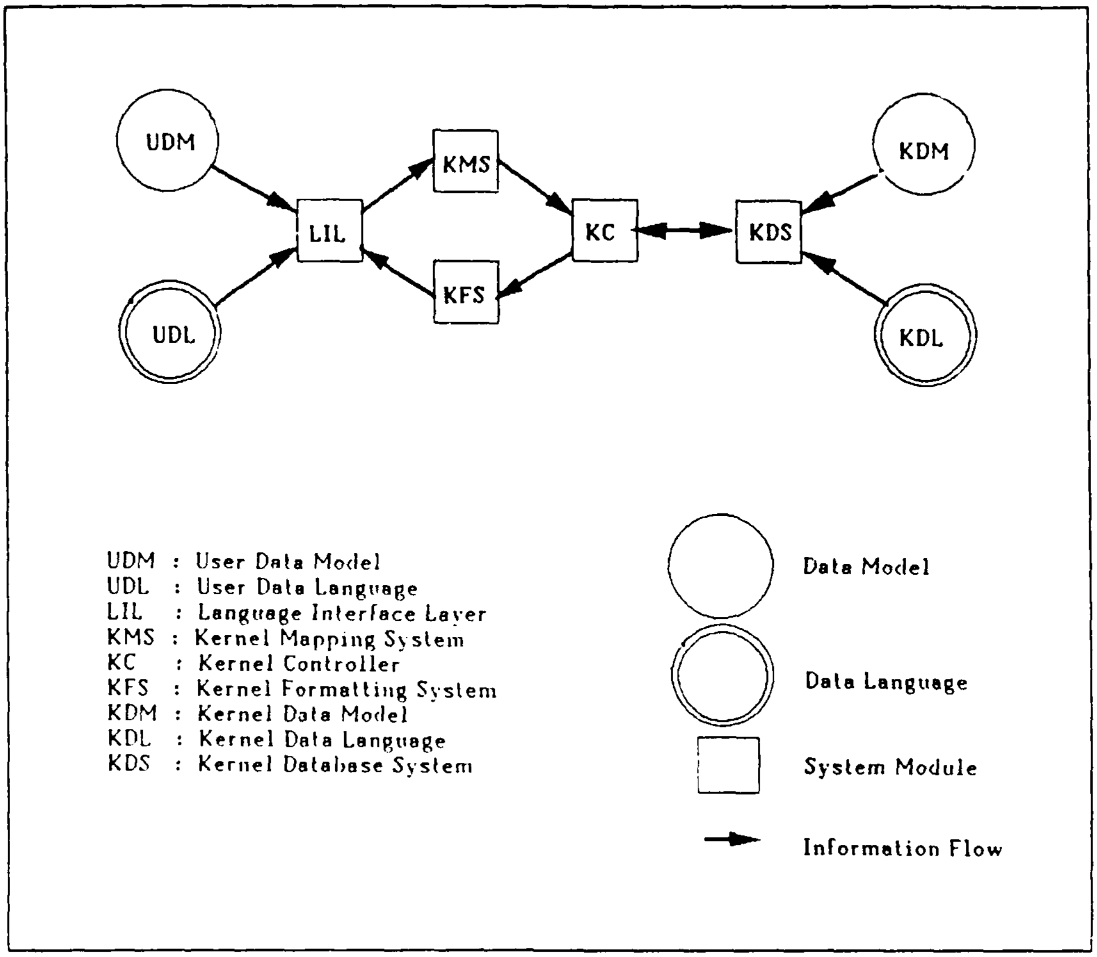


Figure 1.1 The Multi-Lingual Database System

the kernel formatting system (KFS). KFS transforms the results from the KDM structure to UDM equivalent via LIL. The user is then informed that the request has been processed and more requests can be accepted.

If the transaction specified by the user is a database manipulation request, KMS translates UDL transaction to KDL equivalent and sends KDL transaction to KC. KC sends KDL transaction to KDS for execution. Once processing is complete, KDS sends the results in KDM form to KFS, via KC, for transforming from the KDM form to the UDM form. The KFS then returns the results of the transformed data to UDM.

The LIL, KMS, KC and KFS are collectively known as the language interface. Four similar modules are required for each language interface of the MLDS. For example, there is a separate and unique set of LIL, KMS, KC and KFS for each model. Currently, these models and their corresponding languages are relational/SQL, hierarchical/DL/I, network/CODASYL-DML and functional/Daplex. On the other hand the KDS is a single, major component shared and accessed by all various language interfaces. Figure 1.2 depicts this concept. It is through the KDS that the actual raw data is accessed and manipulated by the various user-defined language interfaces.

The attribute-based data model and the attribute-based data language (ABDL) have been selected and implemented as the KDM and KDL, respectively, for MLDS. A series of reports show how the relational, hierarchical, network and functional data can be transformed to attribute-based data while at the same time presenting preliminary work on the corresponding data-language transactions [Refs. 5,6,7]. More recent works provide a complete set of algorithms for the data-language translations from SQL to ABDL [Ref. 8], from DL/I to ABDL [Ref. 9], from CODASYL-DML to ABDL [Ref. 10], and from Daplex to ABDL [Ref. 11]. Additionally, the language interface software has been completed for relational [Ref. 12], hierarchical [Ref. 13], and network [Ref. 14] data models. The language interface software for the functional model has not been completed at the time of this thesis, but detailed design for its implementation has been documented [Ref. 15]. Additionally, there is also research into incorporating a language interface for an object-oriented data model and its specified data language. The model is the Graphic LAnguage for Database (GLAD) model using the Actor language currently under development at the Naval Postgraduate

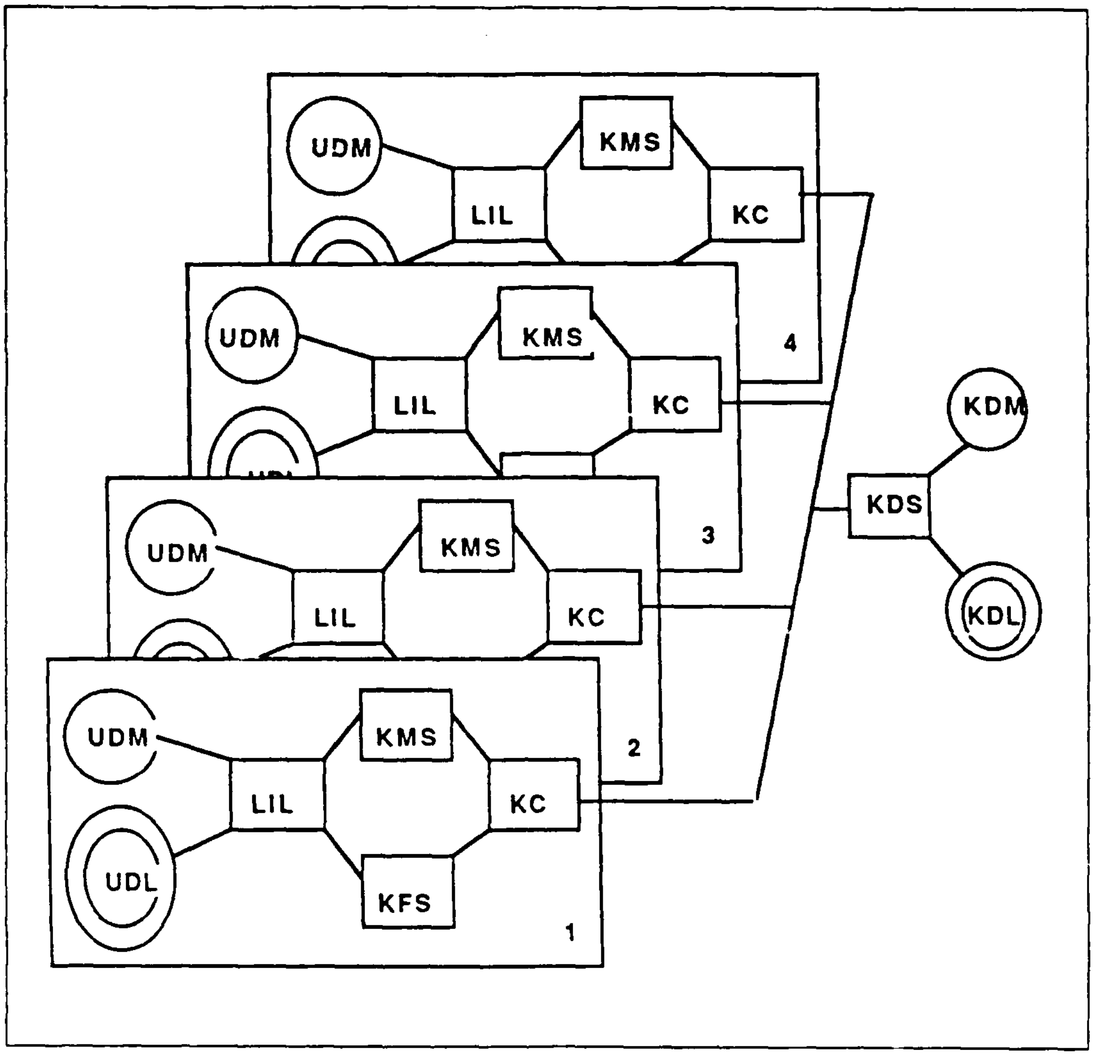


Figure 1.2 Multiple Language Interfaces for the Same KDS

School. Associate Professor of Computer Science Thomas C. Wu is in charge of this project.

## 2. The Multi-Backend Database System

To overcome the performance and upgrade problems associated with the traditional approach to database system design, the multi-backend database system (MBDS) was designed. MBDS has solved these problems by using multiple backend processors connected in parallel to a single controller. Each backend has its own hardware, software and disk system, as shown in Figure 1.3. Hardware and software are not unique to each backend, but is replicated over all backends. The backend processors are connected to the backend controller via a communication bus. The backend controller can be accesses either by the user directly or through the host computer. The backend controller is responsible for supervising the execution of database transactions.

Performance gains are realized by increasing the number of backend processors. If the database size remains constant, then the response time for the user transaction is inversely proportional to the number of backend processors of the system. Also, if the number of backend processors increases in direct proportion to the database size, then MBDS produces nearly invariant response time for the same transaction. For a more detailed discussion on the MBDS the reader is referred to [Refs. 16 and 17].

## C. THESIS ORGANIZATION

Under the current mode of operation there is no efficient manner in which to generate or reorganize large databases. Current operations are satisfactory for single record insertions. The process to actually determine where a record should be inserted into the database is quick and simple. Each insert operation generates a set of I/O operations. This in itself is not bad, but when performed several thousand times, it can take several days to load a new and large database.

When the delete operation is used, records are not physically removed from the database, but are actually tagged for deletion. Over a period of time, these tagged records grow in numbers. They should be periodically collected and removed from the

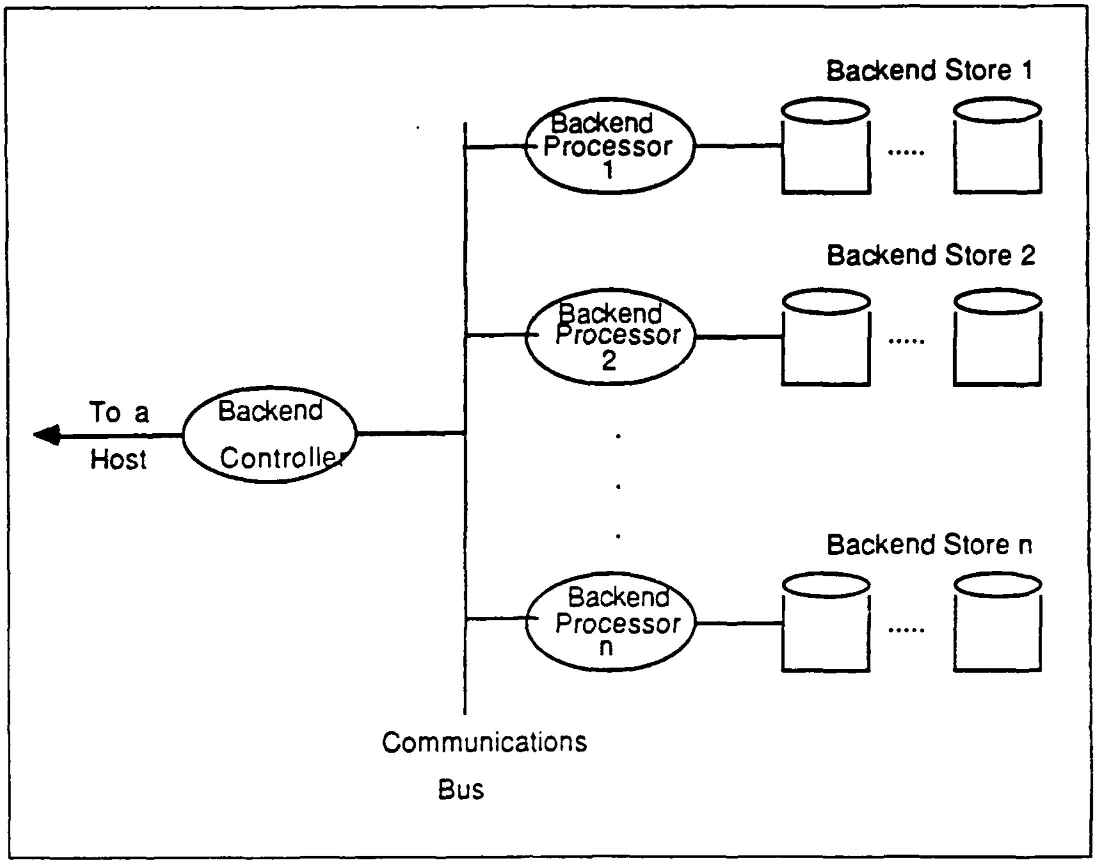


Figure 1.3 The Multi-Backend Database System

database to obtain the maximum used of disk space for active records. This thesis will provide an algorithm which is flexible enough to create large databases and provide a "garbage collection" feature to get rid of the tagged records and reorganize remaining active records.

In Chapter II, we look at the kernel software and describe the system's limitations in regards to creating new databases and deleting records from a database. We also describe two approaches for an algorithm designed to create and/or reorganize large databases over multiple backends, then select the appropriate approach. Chapter III outlines the multiple phases of the algorithm. A complexity analysis of the new algorithm versus the complexity analysis of the insert operation is given, indicating the efficiency of the new algorithm. Also included in this chapter is a pseudo-code design of the actual algorithm. In Chapter IV, we discuss implementation issues. In Chapter V, we make our conclusions about the proposed design. Finally, Appendices A, B, C, and D provide the program design specifications for building the Attribute Table, Descriptor-to-Descriptor-Id Table, Cluster Definition Table and program specifications for the Record Error Checker routine, respectively.

# II. THE KERNEL SYSTEM

In this chapter we discuss the overall configuration of the kernel system. In the first section, we discuss the kernel data language and the kernel data model. In the second section, we discuss the specifications for the input files (i.e., template, descriptor and record). In the third section, we discuss the limitations of the system with respect to database creation and garbage collection. In the final section we discuss possible solutions and select a proposed solution for overcoming the limitations discussed in the second section.

## A. THE KERNEL DATA MODEL AND THE KERNEL DATA LANGUAGE

The kernel system is composed of two parts: the kernel data model and its model-based data language. The kernel data model used in the multi-backend database system (MBDS) is the attribute-based data model. The kernel data language that supports the attribute-based data model is the attribute-based data language (ABDL). The next two sections introduce the concepts and terminology of the kernel system.

### 1. The Attribute-Based Data Language

ABDL supports five primary database operations: INSERT, DELETE, UPDATE, RETRIEVE and RETRIEVE-COMMON. A request in ABDL is a primary operation with a qualification. A qualification is used to specify the part of the database on which to operate.

The INSERT command is used to insert a new record into the database. The qualification of the INSERT request is a list of keywords and a record body being inserted. For example, the following INSERT command

```
INSERT(<FILE,USCensus>,<CITY,Cumberland>,<POPULATION,4000>)
```

will insert a record into the US Census file for the city of Cumberland with a population of 4,000.

The DELETE request is used to remove one or more records from the database. The qualification of a delete record is a query. A query, in the DELETE operation, specifies which record in the database will be deleted. For example, the following DELETE command

```
DELETE ((FILE = USCensus) and (POPULATION > 100000))
```

will delete all records from the US Census file with a population greater than 100,000.

The UPDATE request is used to modify records of the database. The qualification of the UPDATE request consists of two parts: the query and the modifier. The query specifies which records of the database are to be modified and the modifier specifies how the record to be updated will be changed. For example, the following UPDATE command

```
UPDATE (FILE = USCensus) (POPULATION = POPULATION + 5000)
```

will modify all records in the US Census file by increasing all populations by 5,000. In this example the query is (FILE = USCensus) and the modifier is (POPULATION = POPULATION + 5000).

The RETRIEVE request is used to retrieve records in the database. The qualification of the RETRIEVE request consists of three parts: the query, a target list, and a by-clause. The query identifies the record to be retrieved, and the target list consists of the attributes (fields name in the record) whose values are to be output to the user. The by-clause, which is optional, is used to group records. Also, the RETRIEVE command may consist of an aggregate operation (i.e., COUNT, SUM, AVG, MIN, MAX) on one or more output attribute values. For example, the following RETRIEVE command

```
RETRIEVE ((File = USCensus) and (POPULATION > 50000)) (CITY)
```

will output the city of all records in the US Census file with populations greater than 50,000.

The RETRIEVE-COMMON request is used to merge two files by common attribute value. The qualification of the RETRIEVE-COMMON request consists of three parts: the query, the target list, and a common attribute between the two files. The query and target list are used as specified above, while the attribute is the key to join the two files. The RETRIEVE-COMMON command in ABDL functions the same as the JOIN command in SQL. For example, the following RETRIEVE-COMMON command

```
RETRIEVE ((FILE=CanadaCensus) and (POPULATION>50000)) (CITY)
COMMON (POPULATION, POPULATION)
RETRIEVE ((FILE = USCensus) and (POPULATION > 50000)) (CITY)
```

will find all records in the Canada Census file with a population greater than 50,000, find all records in the US Census file with a population greater than 50,00O, identify the records from these files whose populations figures are common and return the city names whose cities have the same population figures.

With these five simple commands, ABDL provides the user with the means to access and manipulate the database.

### 2. The Attribute-Based Data Model

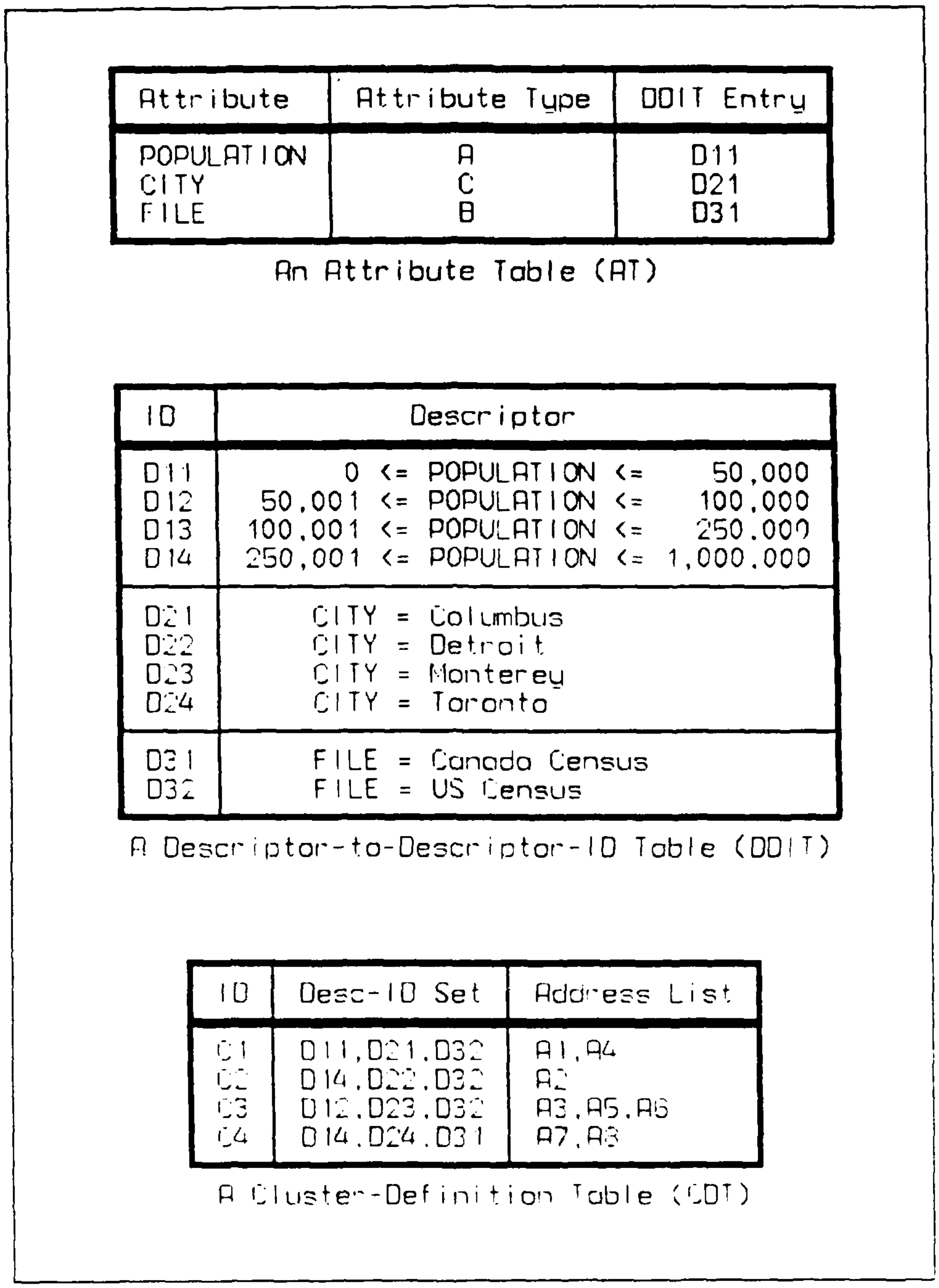
In the attribute-based data model, data is considered in the following constructs: the database, files, records, attribute sets, value domains, attribute-value pairs, attribute-value ranges, keywords, directories, directory keywords, non-directory keywords, keyword predicates, record bodies, and queries. These constructs are applied to two types of data: meta data and base data.

#### *a. The Meta Data*

The meta data is the stored data about the structure and form of the base data. The various meta data constructs form the directory of the database. The directory contains the following constructs: attributes, descriptors, and clusters. The attribute is used to represent a category of certain common property of the base data. A descriptor is used to describe a unique range of values or distinct value for the attribute. A cluster is a group of records in which every record in the cluster is of the same set of descriptors. More specifically, the directory is organized in three tables: the attribute table (AT), the descriptor-to-descriptor-id table (DDIT), and the cluster-definition table (CDT). AT maps directory attributes to descriptors, while DDIT maps each descriptor to a unique descriptor-id. CDT maps descriptors-id sets to clusters-ids. There are three classifications of descriptors. A type-A descriptor is a conjunction of less-than-or-equal-to and greater-than-or-equal-to predicates, where the same attribute appears in each predicate. A type-B descriptor consists of equality predicates. A type-C descriptor defines a set of type-C sub-descriptors. The type-C sub-descriptor are equality predicates defined over all unique attribute-values which exist in the database. A sample of each directory table is provided in Figure 2.1.

### *b. The Base Data*

The base data is the actual raw data that makes up the database. A database is a collection of files. These files contain a group of records that are related by a unique set of directory keywords. A record is composed of two parts: an attribute-value pair (or keyword) and the textual information (record body). An attribute-value pair is a member of the Cartesian product of the attribute name and the value domain of the attribute. For example, <POPULATION, 50000> is an attribute pair having 50,000 as the value for the population attribute. A record contains at most one attribute-value for each attribute defined in the database. Certain attribute-value pairs of a record or file are called directory keywords and are kept in the directory for



| Attribute | Attribute Type | DDIT Entry |
|---|---|---|
| POPULATION | A | D11 |
| CITY | C | D21 |
| FILE | B | D31 |

An Attribute Table (AT)

| ID | Descriptor |
|---|---|
| D11 | 0 <= POPULATION <= 50,000 |
| D12 | 50,001 <= POPULATION <= 100,000 |
| D13 | 100,001 <= POPULATION <= 250,000 |
| D14 | 250,001 <= POPULATION <= 1,000,000 |
| D21 | CITY = Columbus |
| D22 | CITY = Detroit |
| D23 | CITY = Monterey |
| D24 | CITY = Toronto |
| D31 | FILE = Canada Census |
| D32 | FILE = US Census |

A Descriptor-to-Descriptor-ID Table (DDIT)

| ID | Desc-ID Set | Address List |
|---|---|---|
| C1 | D11,D21,D32 | A1,A4 |
| C2 | D14,D22,D32 | A2 |
| C3 | D12,D23,D32 | A3,A5,A6 |
| C4 | D14,D24,D31 | A7,A8 |

A Cluster-Definition Table (CDT)

Figure 2.1 Sample Directory Tables

identifying records or files. Those attribute-value pairs not kept in the directory are called non-directory keywords. A record in the files of the database constitutes the base data of the database. Below is an example of a record.

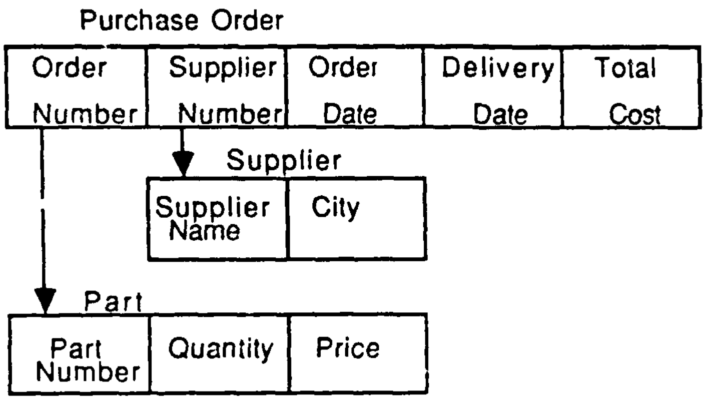
(<FILE,USCensus>,<CITY,Marina>,<POPULATION,50000>,{Moderate Climate})

Angle brackets, <,> enclose the attribute-value pair, curly brackets, {,}, enclose the record body, and the record itself is enclosed in parenthesis. The records of the database may be identified by keyword predicates. A keyword predicate is a three tuple consisting of a directory attribute, a relational operator (i.e., =,!=,<,>,<=,>=), and an attribute-value. Combining keyword predicates in disjunctive normal form characterizes a query of the database. The query specifies which record in the database is to be accessed and/or modified.

## B. THE TEMPLATE SPECIFICATION

The construction of a database is controlled by three input files of the database: the template file, the descriptor file, and the record file. The template file defines the directory and record structures of a file. The descriptor file contains the directory attributes and their descriptor definitions. The record files contain the actual data or records. These files are used by MBDS to form the record clusters.

To better describe the input files created, consider a purchasing system within some large business. The system consists of purchase-order, part, and supplier records. Figure 2.2 describes the relationship of the records. Figure 2.2.a shows the record schema and Figure 2.2.b shows how this schema would be normalized. Each record shows the name of each template (file) and its attributes. The process of normalization captures the data relationship from one file to another, as well as, provides a form of representation that is suitable for the template specification. Normalization requires that



(a) Schema for a Purchase Order System

Purchase-Order (order-number, supplier-number, order-date, delivery-date, total-cost)

Part (order-number, part-number, quantity, price)

Supplier (supplier-number, supplier-name, city)

(b) Normalized for a Purchase Order System Schema

Figure 2.2 Sample Record Relationship

certain attributes appear in more than one file. The supplier number in the purchase-order file is repeated in the supplier file and is combined with the supplier name to form a unique identifier. This duplication does not imply that the value is redundantly stored because normalization is concerned with logical structures rather than physical organization

### 1. Template Descriptions

The template file contains the descriptions of the templates defined in the database. In general, there may be many databases in the MBDS. So keeping this in mind, the template descriptors for the records in the different databases must be kept separate. The format of a template file for a given database with n templates could be as follows:

Database name
Number of templates in the database
Template description for template #1
Template description for template #2
.
.
Template description for template #n

A typical descriptor with m attributes could be as follows:

Number of attributes in a template
Template name
Attribute #1 data type
Attribute #2 data type
.
.
Attribute #m data type

There are three different data types: integer (i), string (s), and floating point (f). The name of this database will be PURCHASING. The template names will be Purchase-Order, Part, and Supplier. With this information we can now generate the template file for the PURCHASING database, as shown in Figure 2.3.

### 2. Descriptor Specifications

A descriptor is a keyword predicate of the form, for example, (supplier-name = WANG) or (price = \$200). MBDS recognizes two kinds of keywords: non-directory keywords for search and retrievals, and directory keywords for search and

```
PURCHASING
3
6
Purchase-order
template          s
order-#           s
supplier-#        s
order-date        s
delivery-date     s
total-cost        f
5
Part
template          s
order-#           s
part-#            s
quantity          i
price             f
4
Supplier
template          s
supplier-#        s
supplier-name     s
city              s
```

Figure 2.3 Sample Template File

retrievals as well as forming clusters. The descriptor file is an input file that contains directory keyword descriptors only. Cluster formulations are based on the attribute values and value ranges of the descriptors. As an example, a cluster containing records in the purchasing database for purchase orders from a supplier WANG with a total cost of $10,000 and up to $50,000 since 1988, is derivable with a set of three descriptors: (supplier-name = WANG), (10,000 =< total-cost < 50,000), and (order-date = 1988).

There are three types of descriptors: type-A, type-B, and type-C. A type-A descriptor is a conjunction of two predicates: less-than-or-equal-to and greater-than-or-equal-to. An example of a type-A descriptor is (10,000 =< total-cost < 50,000).

For creating a type-A descriptor, the attribute (i.e., total-cost) and the value range (i.e., of 10,000 and up to 50,000) must be specified. The value range is expressed in terms of upper and lower limits. A type-B predicate is an equality predicate. An example of a type-B descriptor is (supplier-name = WANG). The type-C predicate is also an equality predicate. However, its values are provided later by the input record. These descriptors are then automatically converted to a set of type-B descriptors with the same attribute name and values corresponding to the value range. As an example, if the template has a type-C descriptor with values of purchase-order, part, and supplier provided by the record, the first set of type-B descriptors generated is (template = purchase-order), (template = part), and (template = supplier).

When specifying descriptors, the attributes of the given descriptor must be unique and the specification of the values and the value ranges must be mutually exclusive. Below is an example of a descriptor file with n descriptors.

Database name
Descriptor definition #1
Descriptor definition #2
.
.
Descriptor definition #n
$

The "$" indicates the end of the descriptor file. Each descriptor definition is expressed in terms of the attribute and its associated descriptor type and data type and followed by the value ranges as shown below.

Attribute Descriptor-type Data-type
Value range 1
Value range 2
.
.
Value range k

@

Value ranges are expressed in terms of upper and lower limits. Since type-B and type-C are equality values, which are exact. The placehold for the upper limit is used for holding the exact value. The lower limit is not used. The value of the lower limit is therefore indicated by an "!". The "@" indicates the end of the descriptor file. An example of a descriptor file is depicted in Figure 2.4.

### 3. Database Records

Once the template and descriptor files are defined, the data record format will be specified. Data records can be prepared in separate files for loading. The format for a typical record file is as follows:

```
Database name
@
Template name #1
Record #1 template #1
Record #2 template #1
      .
      .
Record #n template #1
@
Template name #2
Record #1 template #2
Record #2 template #2
      .
      .
Record #n template #2
      .
      .
$
```

The database name identifies the database to which the template and the record belongs. The "@" indicates the beginning of a new template followed by the template's name. All records following the template name belong to that template until

```
PURCHASING
template          C                 s
!                 Purchase-order
!                 Part
!                 Supplier
@
total-cost        A                 f
1000.00           100000.00
100000.00         500000.00
@
order-#           A                 s
#1                #50
#50               #100
#100              #1000
@
price             A                 f
1000.00           50000.00
50000.00          500000.00
@
supplier-name     B                 s
!                 WANG
!                 IBM
!                 DEC
@
city              B                 s
!                 Monterey
!                 San Jose
!                 Carmel
@
$
```

Figure 2.4 Sample Descriptor File

either the "@" or "$" is encountered. The "$" indicates the end of the entire record. Values in the record are separated by a space. An example of a record in the Purchasing database in the purchase-order template would look like

<order-#,26>,<supplier#,51>,<order-date, 18-May-1988>

<delivery-date,22-Jul-1988>,<total-cost, 200,500.00>

If we extract the values of each attribute value pair, we have

26 51 18-May-1988 22-Jul-1988 $200,500.00

which is a record of the purchase-order template in the Purchasing database. Figure 2.5 gives a more complete record file.

## C. LIMITATIONS

The kemel system described in the earlier section provides its users with the same functional capabilities as most other database systems. It provides its users with a clean, easv way to access and manipulate data. It is a simple, but powerful database system. Yet for a!! its simplicity, the system lacks an efficient means to generate very large databases.

The capability to create large databases does exist but it is slow, cumbersome, and quite tedious. The INSERT command is the means by which records are inserted into the database. Let us briefly discuss how this process works.

In order to add a record to the database an insert operation is issued by the user. The kemel system receives the command, recognizes it as an insert operation and proceeds to process the request. The record undergoes a record processing phase where it is checked for validity prior to insertion on to the backend. Record processing is a necessary and very involved process and is outlined here.

The INSERT operation requires considerable work on the part of the kemel system. The complication is due to the type of directory keywords of the record to be inserted. If the attributes of the directory keywords are of type-A or type-B, then there are no complications. The complications arise when the attributes of the directory keywords are of type-C. We first look at the steps required to insert a record without type-C attributes.

```
PURCHASING
@
Purchase-order
26   51        18-May-1988    22-Nov-1988    191500.00
31   51        25-jun-1988    14-Apr-1889    381900.00
@
Part
26   V780      VAX-11/780          1          91500.00
26   M780      Memory              8          42000.00
26   D81       RA81                1          58000.00
31   V8600         VAX-8600        1         381900.00
@
Supplier
51   DEC       Carmel
$
```

Figure 2.5 Sample Record File

In step one, the attribute-value pairs (or keywords) of the record are identified. The attribute, of an attribute-value pair, is used for matching the attributes in AT. A successful match indicates that at least one keyword of the record is a directory keyword. There may be more than one match if the record contains more than one directory keyword. If there is not a match, then the keyword of the record is not a directory keyword. If none of the attributes of a record is a directory keyword, the record for insertion is rejected by the system.

In step two, once the attribute-ids are obtained, the descriptors in DDIT are searched to determine whether a descriptor covers the keyword of the record for insertion having the same attribute. If the descriptor covers the keyword then the corresponding descriptor-id is used to form the descriptor-id group.

In step three, a search for a descriptor-id set in CDT which is identical to the descriptor-id group is performed. If a descriptor-id set is found to be identical to the descriptor-id group, then a cluster has been identified to receive the record to be inserted. If a descriptor-id group is not found during the search of CDT, this implies: either (1) the descriptor-id group is a new descriptor-id set, or (2) the descriptor-id group cannot be a descriptor-id set. In case (1), this indicates that a new cluster is to be formed for the record to be inserted. A new entry in CDT is created for the descriptor-id set and its associated new cluster-id. In case (2), the descriptor-id group cannot form a new descriptor-id set. In this case, the record is rejected.

In step four, we are concerned with the placement of the record into the cluster identified in step three. The record-id is transformed into a disk address and the record is placed at the secondary storage so addressed. This process of record placement is more complicated than it sounds. The records of a cluster must be evenly distributed across the multiple backends, so the placement of the record on a backend is not a simple task. The following steps are taken:

(1) The cluster-id set *is sent to the controller.* A backend then receives from the controller the information on the backend whose disk has the available block for the cluster. This information is obtained from the table known as the cluster-to-next-backend table (CINBT), which is maintained by the controller. In the beginning, the controller does not know the cluster into which the record is to be inserted. The controller tasks all the backends to work on the meta data and to identify the cluster to receive the record by broadcasting the operation and record to all backends, via the communication bus. Meta data processing by backends is parallel operations. One the cluster-id is determined by the backends, the controller receives the same cluster-id from all the backends, again via the communication bus.

(2) The controller locates the entry in the CINBT whose cluster-id is identical to the cluster-id received. Once the cluster is identified, it is checked to determine if enough space is available to receive the record. If space is available, the cluster size is decreased by the length of the record to be inserted. If the space is not large enough, the controller forfeits the space and allocates a new block of secondary storage from the next backend. The controller then sends a message over the communication bus, simply instructing the backend to write the record into the backend's available secondary space. To write a record we retrieve the backend's portion of the cluster from secondary storage, write the record into that portion and then stores the cluster portion back onto the secondary storage. We note this operation is not parallel since there is only one backend performing the writing.

The entire process described above is very efficient when inserting a single record, but not when creating a very large database consisting of thousands or millions of records. Each record generates two I/O requests and the hardware is idle during I/O processing. Thousands of records will generate thousands of separate I/O requests. Thus, we can see where the inefficiency lies when using the current mode of operation to create very large databases. What we attempt to do in this thesis is to minimize the involved process on the record-by-record basis where the entire kernel system is tied up by the large number of insert operations for a large number of records.

The kernel system lacks the capability to reclaim storage space once a record has been identified for deletion. The DELETE command does not physically remove a record from the database. The DELETE command simply tags the record, identifying it as no longer active. The system no longer recognizes this record as a part of the database, but the record is still a physical part of the database since it still continues to reside on secondary storage. What now resides on secondary storage are active records and "garbages" (i.e., tagged records). This results in fragmentation within the storage medium. If all records were of the same size then a new record added to the database would fit nicely into the space occupied by the tagged record and there would be no fragmentation. Unfortunately, in most databases not all records are fixed-length. Some are variable-length. If the new record to be added to the database is smaller than the record that previously lived at that spot, then the new record could be inserted into this space. But a smaller record will not solve the problem of fragmented space on disk. It only creates smaller fragments. If the new record to be inserted is larger than the previous record, then it cannot be inserted into that slot and must be placed someplace else on the medium. When most large systems want to reclaim memory that is no longer needed, they perform an operation called storage reclamation or compaction. This type of function involves gathering occupied areas of storage into one end or the other in secondary storage. This leaves a single large free storage hole instead of numerous small holes. This concept can be applied to the database problem where records tagged for deletion are not required to remain in the database. If all the active records are collected and then redistributed onto secondary storage, this will rid the storage medium of fragmentation. In other words, the reorganization of secondary storage is the creation of a new database with only active records. Compressing the active records remaining in storage is the route that should be taken.

## D. POSSIBLE SOLUTIONS

In this section we propose two solutions to resolve the problem for the lack of an efficient database creation and reorganization function. The basic idea behind each solution is to bypass the system and to load the data directly to the disks. With both proposed solutions, the intention is to speed up the process of creating very large databases. After each proposal is presented, the best possible solution will be selected. Both approaches use the same type of input and will produce the same output. The inputs to the algorithm are: (1) the template file, descriptor file, and a record file (on tape medium) if creating a new database or, (2) the AT and DDIT of the database to be reorganized along with a record file (on tape medium) containing the data to be redistributed. The output, once the algorithm is used, will generate a database over multiple disks consisting of both base and meta data.

### 1. One-Pass Approach

The One-Pass approach is so named because of the number of times each record will be handled before it is placed onto secondary storage. The basic strategy for this algorithm is as follows:

(1) Generate the AT and DDIT

(2) Load records from tape into temporary work space

(3) Process each record - check syntax and format record

(4) Update DDIT - adding new Type-C attributes

(5) Build CDT - if first record on first track/ Update CDT

(6) Assign records to cluster

(7) Write cluster to backend

(8) Repeat steps three through seven in processing each record

(9) Write meta data to secondary storage

The advantage to this algorithm is that each record is handled only once. The disadvantage to this algorithm is that clusters of records are written onto the secondary storage many times as the clusters grow in size.

### 2. Two-Pass Approach

The two-pass approach informs us that data will be handled twice before it will be placed onto secondary storage. The basic strategy for this algorithm is as follows:

```
Phase-One
(1) Generate AT and DDIT
(2) Load records into a temporary work space
(3) Process each records - check record syntax and format record
(4) Update DDIT
(5) Build/Update CDT
(6) Repeat steps three through five in processing all records
(7) Determine record distribution
    If good distribution and/or small percent of bad records then
        go to Phase-Two
    else redefine descriptor file and repeat Phase-One.

Phase-Two
(1) Scan and sort records - by each cluster-id number
```

(2) Build CINBT

(3) Load records onto their disks

(4) Write meta data to their disk

(5) Send CINBT, IIGAT and IIGDDIT to controller disk.

The advantages to this approach are: (1) it enforces even distribution of records among the clusters, (2) it sorts records by cluster-id number, which assists in the building and loading of clusters of records onto disks, (3) it disallows proceedings into the second pass if a large percentage of the input records are corrupted and (4) it writes clusters of records only once onto the permanent storage. The disadvantage to this algorithm is that each record is handled twice. IIGAT and IIGDDIT are insert-information-generation-attribute-table and insert-information-generation-descriptor-to-descriptor-id-table, respectively. These tables are part of the meta data residing on the controller and are used for generating new type-C descriptors. IIGAT houses the attribute and the next DDIT entry for a new type-C descriptor. IIGDDIT houses all the type-C descriptor values. These two tables are used to generate new type-C descriptor-ids for pre-defined type-C attributes.

### 3. The Selected Solution

In selecting the best solution to the database creation and reorganization problem, we must weigh the advantages and disadvantages of each proposed solution. As it appears in our case, the advantage of one approach is the disadvantage of the other. So based on this, we must determine which approach will yield to us the most effective and efficient manner in which to store large databases. The two major advantages mentioned are how often a single record is handled, getting a good record distribution within the clusters and how often a cluster is written onto the secondary storage. We look at these advantages in more detail and determine the relative importance of each.

In the One-Pass approach the record is processed and then placed onto secondary storage. In the Two-Pass approach the record is processed in pass-one and then, during pass-two, it is read, sorted and placed onto secondary storage. The record in the two-pass approach is read/handled twice. It is important to understand why the record is handled twice, and because it is handled twice, what impact, if any, it has on the overall efficiency of the algorithm.

As mentioned earlier, the use of the INSERT command is the only way in which a record is added to a database. Single record insertion is quick and easy. The problem arises with large database creation, when each record is individually inserted using the INSERT command. We stated that the problem is encountered because of the large number of I/O operations that must be performed. If we look at the general design of the two algorithms, we see in the One-Pass approach that each record processed generates an I/O operation for writing the record onto the secondary storage. This is similar to the current design and no performance gain is encountered. In the Two-Pass approach each record does not generate an I/O request. The placing of the records onto secondary storage is deferred to the second pass where records are written as clusters of records and not individually. Since records are written in clusters vice each record generating its own I/O request, the number of I/O requests is reduced greatly.

Since records of a cluster are available for distribution on the parallel disks, an even distribution of records within a cluster will produce better disk utilization and speedier data retrieval. The numbers of records within clusters may vary greatly. To facilitate even distribution, we should choose smaller blocks sizes, say one-half or one-quarter track. On the other hand, smaller block sizes may have difficulty to accommodate large record sizes, since we do not split a record over different blocks which are of two different backend's disks, respectively.

We have shown that the number of times a record is handled does not necessarily imply the processing will be slower. The One-Pass approach writes each cluster of records repeatedly onto the disks as the cluster grows. This will require a new distribution of the records of the cluster which induces many computations and I/O operations. We note it is easier to have a good record distribution among clusters when all records of a given cluster are ready for distribution. Since the Two-Pass approach embodies both of these concepts, it is our choice as the best approach. These concepts are (1) fewer I/O operations for cluster creation, and (2) better distribution of clustered records on parallel disks due to one-time record collection and distribution.

# III. THE PROPOSED DESIGN

In this chapter we discuss the algorithm in detail. Specifically, we look at the multiple phases of the Two-Pass approach. Also in this chapter, we provide a time-complexity analysis for this algorithm to show that it does provide a more efficient means, over current operations, of generating large databases. Finally, we provide a Pascal-like pseudo code of the actual design specifications for the algorithm.

## A. THE DESIGN OF DATABASE CREATION/REORGANIZATION

The INSERT command is used to insert a single record into the database. In order to create a large database holding thousands or millions of records, under the current mode of operation we must execute as many INSERT commands as there are records to be inserted. Although it is possible to generate a large database under this method, it is a very laborious and inefficient process.

The DELETE command is used to delete records from the database. Records are not physically removed from the database but are actually tagged as no longer active. These tagged records are considered "garbages" in the database and lead to fragmentation on the medium. To rid the medium of this fragmentation, we collect all the active records and then redistribute them evenly by clusters across the multiple disks. We discovered in the previous chapter that ridding the system of its tagged records is in essence creating a new database with its active records.

We have designed an algorithm which is flexible enough for both database creation and reorganization (garbage collection). The user will specify in the beginning stage what type of operation he/she will be performing. Based on this input, the algorithm will proceed to perform either a creation or reorganization function. The input, output, and internal processing for the type of operations to be performed are slightly different, but in either case this algorithm will produce a database that is evenly distributed across the multiple backends per cluster.

## B. MULTIPLE PHASES OF THE ALGORITHM

The proposed algorithm has two phases: Phase-One and Phase-Two. The first phase is concerned mostly with record processing, while the second phase is concerned with loading data, both meta and base, onto the backends. The logic used in the two-phase approach is similar to the logic used by MBDS. The difference between them is when the record is actually placed on disks. The next two sections will discuss in detail the Two-Phase approach.

### 1. Phase-One

Phases-One is concerned primarily with record processing and building the three main directory tables: the attribute table (AT), the descriptor-to-descriptor-id table (DDIT) and the cluster-definition table (CDT). Upon entering this phase, the user must have specified the type of function he/she wishes to perform (i.e., create a new database or reorganized active data). If the user has specified a creation of a new database then the template and descriptors files for the database will be retrieved. Once these files are retrieved, the AT will be generated. A partial DDIT will also be built. AT maps directory attributes to descriptors and DDIT maps each descriptor to a unique descriptor-id. The descriptor file contains all directory keyword descriptors.

Each descriptor definition is expressed in terms of the attribute, its associated descriptor type, data type, and followed by the value ranges. It is from this file that the information is obtained to construct both AT and DDIT. To construct AT all attributes and their corresponding attributes types are extracted from the descriptor file. The descriptor file is read in a single pass and when complete all directory keywords will have been extracted and placed into AT. Each keyword is assigned a unique DDIT entry value. This value is used as an index into DDIT to help form the descriptor-id set. At the same time AT is being constructed, DDIT is also being built. Each attribute is followed by its value ranges. Once an entry for the attribute is made in AT its corresponding range values can then be place in DDIT. These values are read sequentially from the descriptor file. The end of the group of range values is reach when the "@" is encountered. Once the descriptor has been read AT and a partial is constructed. DDIT is not completed at this time because all type-C descriptors are not known since they are introduced by the records. If the user has specified a reorganization of the active records (i.e., garbage collection), then AT and DDIT will be retrieved. Since this is a reorganization of active records, their AT and DDIT already exist. Once DDIT is retrieved the type-C attributes will be removed from DDIT. This is to ensure that DDIT contains only attributes that are active. The reasoning behind this is once records are tagged for deletion it is conceivable that all type-C attributes could have been deleted. If not, they may be re-introduced when we encounter them in the active records. Appendices A and B are attached and provide the program specification for building AT and DDIT, respectively.

Once we have built AT and a partial DDIT we can now proceed with the record processing phase. At this point, records are processed the same, whether the function being performed is creation or garbage collection. We start by loading a block of records to be processed from the record file into main memory (or a work space). We then get a single record from our work space, keeping a running total of records in the database. This number will be used later when we determine the percentage of bad records processed. At this point the record processing phase begins. A record is scanned. After scanning the record, it will be checked for errors. First, the record is check to determine its syntax. The first attribute of the record is examined to determine the record template. Once the template is identified as valid, the remaining attributes are checked to determine if the record contains the correct number of attributes and that each attribute also has its correct associated data types. These records are checked against specific data structures. These data structures hold the record template descriptions for all records in the database. These data structures reflect the information contained in the template file. If the record is found to contain errors, it will be sent to a record handler for processing bad records. A count of the total number of bad records will be maintained. If the record is without errors it will be formatted at this time for subsequent placement onto the disk in Phase-Two. Formatting involves embedding special characters within the record. The "#" is the special character which is placed between each attribute in the record and the "&" is the special character used to mark the end of the record. DDIT will be updated and CDT will be built (or new entries will be placed in the table). The CDT built at this time is not a complete CDT because the address of each cluster is not known. Recall that records are not placed onto secondary storage until Phase-Two. The partial CDT provides us with a count for the records in each cluster. This information will be used later when we determine whether or not a good record distribution within the clusters has been achieved. After an entry is made into CDT, the record is then placed into a temporary work space. This entire procedure is repeated until all the records loaded into the work space has been processed. Once the records are processed, we check to see if there are any more records in the record file. If there are more records in the record file to be processed, then another block of records are read into the work space and the procedure is repeated. This process will continue until all records in the record file are processed. After all records are processed, a tape is produced containing all error-free records that compose the database. Appendices C and D are attached and provide the program specifications for building CDT and the record-error-checking routine, respectively.

Once the record processing phase has been completed we must then determine if we should proceed onto Phase-Two. We proceed to Phase-Two under two conditions: if there is a good record distribution within the clusters, and if the percentage of records containing errors is relatively small (e.g., less than 10% of total records processed). We define a good record distribution within a cluster as clusters that are relatively the same size. The ideal size of a cluster would be approximately the number of backends times the track size. During the record processing phase we built CDT. We can now look at the partial CDT, and judge whether or not we have obtained good distribution of the records. A criterion to determine if good distribution is achieved is to look at the average cluster size. If the average cluster size deviates greatly from the ideal cluster size, then a good distribution was not obtained. If it is determined that the records within the clusters are not distributed evenly, then the algorithm will not proceed to Phase-Two. Cluster formulations are based on the attribute values and values ranges of the descriptors. One possible solution to this problem would be the automatic division of the range attributes. At first glance this might be an appropriate solution. It would guarantee a definite redistribution of records within clusters. Problems may be encountered when dividing range attributes that are non-numeric, or this division may result in clusters that are much smaller than the ideal cluster size. In any case, when even record distribution is not obtained the range attributes should be adjusted. These ranges should be either decreased and reduced in sized or combined to created a larger interval. Adjusting all ranges variables may not be required or desired. The procedures described above can be taken in a number of combinations. The readjustment of range attributes for a particular database should be handled on a case-by-case basis.

The second condition under which we proceed to Phase-Two is if only a small percent (say 10%) of the records contain errors. If for some reason data is corrupted and a large percent (e.g., 30%) of the data processed is bad, then this algorithm will not proceed onto Phase-Two. At the time, the user can reexamine the input data, take whatever actions are needed to correct the problem with the data and then reenter the algorithm. If Phase-One is a success (both good record distribution and small percent of bad records), then the algorithm will proceed to Phase-Two. Figure 3.1 provides a flow-chart diagram of Phase-One.

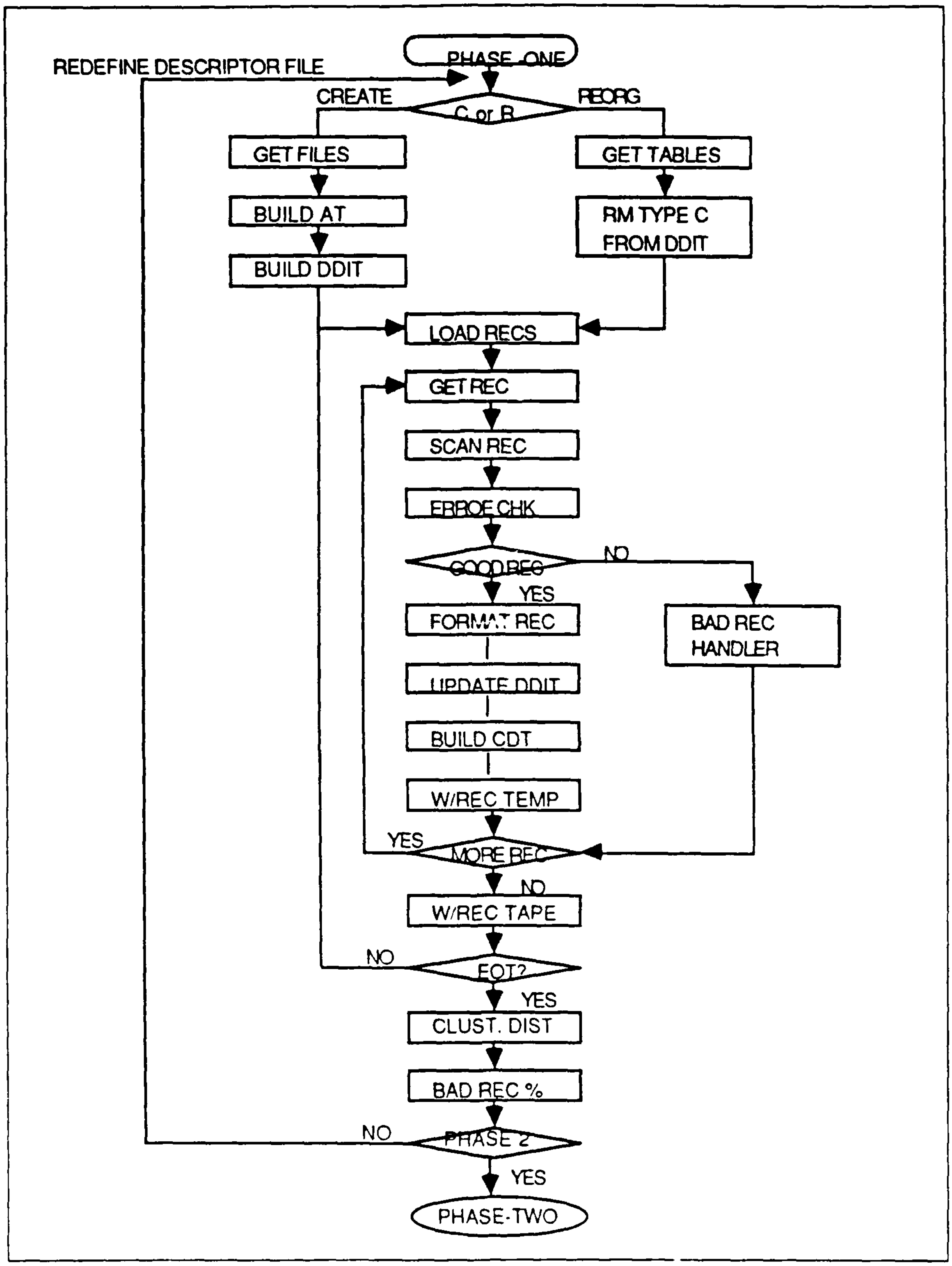


Figure 3.1 Phase-One

### 2. Phase-Two

Phase-Two is mainly concerned with loading the meta data and base data onto disks. The records that make up the database have now been stored on tape. This tape was built during Phase-One. The first function to be performed in Phase-Two is a tape sort. The records on the tape will be sorted by their cluster-id numbers. In other words, all the records whose clusters have the same id will be placed next to each other. Most systems provide a tape-sort capability. There are a number of sort algorithms available (i.e., Quick sort, Bubble sort, etc.) so we will not specify a sort algorithm but leave this decision to the implementor. After the tape has been sorted, a portion of the tape is placed into either main memory or a work space. Since the records are sorted by their cluster-id numbers, the building of the cluster is made simpler. We need only read the records sequentially, since the records of a given cluster are next to each other. When building a cluster we concern ourselves with two things: cluster size and cluster number. Records of a cluster are loaded to disks in blocks. A block will be written to a disk when a sufficient number of records have been collected and the total number of bytes in the records is as close to block (track) size without surpassing it. The first two bytes of each block refer to the total number of bytes in the block (track). Each time a cluster is loaded onto one or more disks the address of the record of the cluster is made into the CDT. The record address, called record-id. therefore consists of the track address and the offset of the record within the track in which the blocked record resides. Each backend has its own CDT. It is at this point that a CDT is being built for each backend. Although all CDTs in the backends have identical cluster definition information (i.e., descriptor-id sets), the ones in their own backends contain only those records-ids that refer to their own tracks. If the record tracks are not on the disks of a backend, the backend's CDT does not have their record-ids in the CDT entries. Also during this loading process, the cluster-id-to-next-backend table (CINBT) is being constructed. The table is part of the controller meta data and this table is used to identify the next backend to receive the next block of records of a given cluster to be placed on the backend's disk. This process is repeated until the entire input tape is processed. After all the records on the tape have been processed, a unique CDT will have been built for each backend. The final tables to be built are insert-information-generation-attribute-table (IIGAT) and insert-information-generation-descriptor-to-descriptor-id-table (IIGDDIT). These tables are also part of the meta data residing on the controller and is used for generating new type-C descriptors. IIGAT houses the attribute and the next DDIT entry for a new type-C descriptor. IIGDDIT houses all the type-C descriptor values. These two tables are used to generate new type-C descriptor-ids for pre-defined type-C attributes. Once AT and DDIT are built, all the information is available to create these two tables. From AT we extract all type-C attributes and from DDIT we get the last descriptor-id of any particular type-C descriptor. We extract this information from AT and DDIT and then insert this data into their respective tables. These tables exist to provide global information on keeping track of type-C attributes which in turn help keep descriptor information on the multiple backends consistent.

All the meta data has been generated and is now ready to be placed on the disks. CINBT, IIGAT and IIGDDIT can be loaded onto the controller disk. If the original function in entering Phase-One was a database creation, then AT and DDIT will be replicated onto each backend and each individual CDT will be loaded to its appropriate backend. If the original function in entering Phase-One was a reorganization of the active records, DDIT will be replicated onto each backend and the individual CDTs will be loaded to their respective backends. Figure 3.2 provides a flow chart diagram of Phase-Two. AT in this function is not affected.

## C. SEQUENTIAL VS. PARALLEL OPERATIONS

Sequential operations infer that all tasks will be performed one at a time, in sequence. No other operation will be performed until the operation before it has been completed. In most cases, sequential processing implies some sort of task dependency. One task cannot start until another task has been completed. Parallel operations infer that one or more tasks can be performed at the same time. The start of one task does not depend on the completion of another task, so these tasks can be processed in unison. In most cases, parallel processing usually provides a quicker response over sequential processing.

The proposed Two-Phase approach algorithm performs sequentially. This algorithm reads a flat file (i.e., the record file). It retrieves "chunks" of records from a file and then processes these records. Once those records are processed, it will retrieve another chunk and continue to repeat this process until all records in the file have been processed. During record processing another flat file is created. This file will contain the records by clusters to be loaded onto the disk space. This clustered record file is also processed sequentially. Records of a cluster are stripped from the tape, blocked into tracks and these blocks are then placed onto backends' disks. The placing of the clusters onto disks is not performed randomly. A round-robin approach

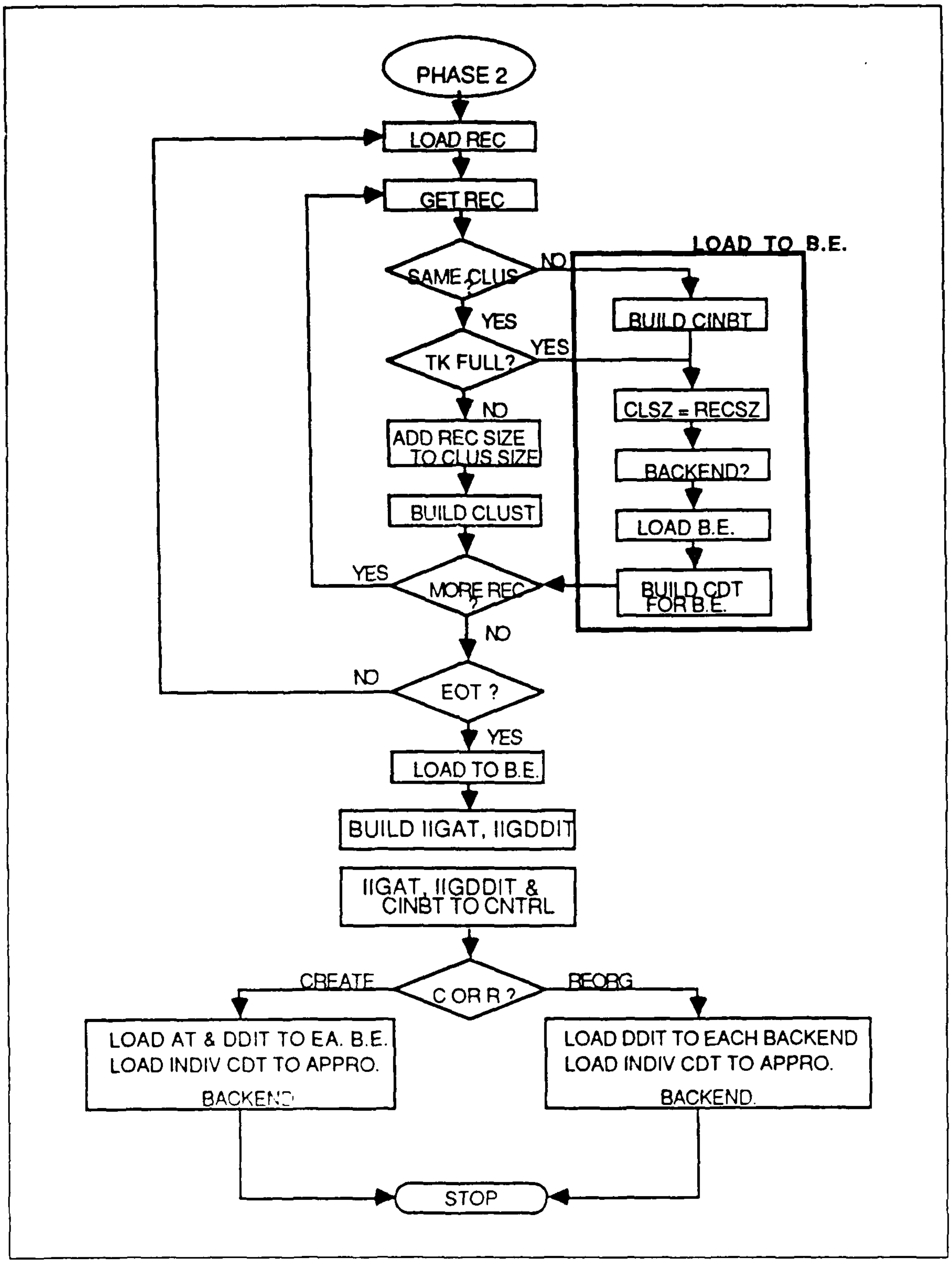


Figure 3.2 Phase-Two

is used. The first backend's disk receives the first block of records of a cluste., the second backend's disk receives the next block of records of the cluster and so on. When the last backend's disk receives its block of the cluster, and if there are more records remaining in the cluster, then the next backend to receive a block of records of the cluster is the first backend's disk again. This procedure is repeated until all records of all clusters are placed on to the disks. There seems to be no overlapping of operations. All tasks are seen to be performed sequentially. Records cannot be processed until they are read into the main memory and they cannot be loaded onto the disks until they are processed. Nevertheless, even though blocking records for given clusters is done sequentially, different blocks of a given cluster may be sent to different backends to be placed on their respective disks in parallel. Here, we have record-serial processing and block-parallel storing operations.

MBDS, when loading records onto the disk, does perform some parallel operations but in a limited capacity. After a record has been assigned a cluster number, the record is ready to be placed on to a backend. The backend controller polls each backends, over the communication bus, searching for the backend on to which the record will be placed. This is where the parallel operations actually take place. All backends will simultaneously search their meta data, specifically their respective CDT, to determine if the record belongs to one of their clusters. All backends will respond to the controller, identifying the cluster to which the record belongs. The controller takes the backends' acknowledgements and identifies the backend to receive the record. The portion of the cluster on the backend is retrieved into main memory, the record is written into that portion of the cluster, and the portion of the cluster is placed back onto the disk. The procedure just described is sequential. When using the INSERT command, the architectural design of MBDS offers very little toward parallel capabilities. The remaining four commands take full advantages of MBDS design to utilize parallel processing to its fullest. If the reader requires more information on the benefits of parallel operations using the other four commands, he/she is referred to the references.

## D. THE TIME-COMPLEXITY ANALYSIS

In this section we present a complexity analysis of the new algorithm versus the complexity analysis of the INSERT operation, indicating the efficiency of the new algorithm.

The new algorithm makes use of the basic strategy behind the INSERT operation. The new algorithm functions almost identically to the INSERT operation. The difference between the two strategies is how the algorithm handles the placement of the data onto multiple disks. Under the current mode of operation for multiple-record-inserts, each record is processed and then generates its own I/O request. This new algorithm on the other hand processes all records, and defers the loading of the records until its second phase. Phase-One will concern itself mainly with the computations that are bound to the CPU, although there are some I/O requests which reads the records from tape into main memory for initial record processing. The second phase will be concerned with operations that are mainly I/O bound.

We will consider the following observations that will be used to simplify our calculations. First, we look at the sequence of operations for both the INSERT command and the new algorithm. Below is the sequence of operations for the INSERT command.

```
load records from the record file
get a record
process a record - check record syntax and format record
update DDIT - new type-C descriptor
update IIGAT and IIGDDIT - new type-C descriptor
update CDT - determine cluster to receive the record
check CINBT -- determine the backend whose available disk
          is to receive the record
I/O -- retrieve cluster to receive record into main memory
write the record to its cluster
I/O -- place the block of the cluster to a specific
          backend's disk
```

Repeat process until all records are processed. The general sequence of operations for the new algorithm is as follows:

```
Phase-One
    load records from the record tape
    get a record
    process record - check record syntax and format record
    update DDIT - new type-C attributes
    update CDT - determine the cluster to receive the record
    repeat until all records are processed

Phase-Two
    sort the record tape by cluster-ids
    load in records from the sorted tape
    build clusters
    update CINBT
    I/O -- write block of records per cluster to specific
              backend's disk
    repeat until all clusters are placed on the backend's disks
    generate IIGAT
    generate IIGDDIT
    I/O -- write AT and DDIT to all backends' meta data disks
    I/O -- write CINBT, IIGAT and IIGDDIT to the controller backend
    I/O -- write different CDTs to different backends'
              meta data disks
```

We can see that both strategies contain similar operations. The major differences are the order in which they are executed and the degree of parallelism in which they have achieved. We pay close attention to where the I/O operations are executed, since this is the area we are seeking to optimize.

Secondly, both the INSERT command and the new algorithm require that AT and DDIT be constructed. Since both strategies use exactly the same process to generate these tables, their initial construction will not be included in the comparison study. We will assume that no I/O operations are required to read the directory tables. These tables will be in main memory at all times. With these considerations, let us proceed with our comparison study. We define the following variables:

| | |
|---|---|
| i: | total number of backends |
| n: | total number of records |
| x: | total number of tracks placed on disks |
| $t_{load}$: | time to read a block of records into main memory |
| $t_{get}$: | time to get a single record |
| $t_{process}$: | time to process a record |
| $t_{ddit}$: | time to inset new type-C attribute into DDIT |
| $t_{cdt}$: | time to build/update CDT |
| $t_{cinbt}$: | time to access CINBT -- determine the backend to receive next the cluster |
| $t_{i/o}$: | time to retrieve/return a cluster from/to a backend |
| $t_{write}$: | time to write record(s) to a cluster |
| $t_{bc}$: | time to build a cluster of track size in 8K bytes |
| $t_{iigat}$: | time to build/update IIGAT |
| $t_{iigddit}$: | time to build/update IIGDDIT |
| $t_{sort}$: | time to perform the tape sort |
| $t_{bcinbt}$: | time to build/update CINBT |

Let us calculate the total time required in the INSERT command to process and load n records into the database. The time required to process a single record through the system is

$$t_{\text{load}} + t_{\text{get}} + t_{\text{process}} + t_{\text{ddit}} + t_{\text{iigat}} + t_{\text{iigddit}} + t_{\text{cdt}} + t_{\text{cinbt}} + 2t_{\text{i/o}} + t_{\text{write}}$$

The time required to process n records through the system is

$$n(t_{\text{load}} + t_{\text{get}} + t_{\text{process}} + t_{\text{ddit}} + t_{\text{iigat}} + t_{\text{iigddit}} + t_{\text{cdt}} + t_{\text{cinbt}} + 2t_{\text{i/o}} + t_{\text{write}}) \text{ -- (1)}$$

We now calculate the total time required, using the new algorithm, to process and load n records into the database. The time required to process n records in Phase-One is

$$n(t_{\text{load}} + t_{\text{get}} + t_{\text{process}} + t_{\text{ddit}} + t_{\text{cdt}})$$

The time required to load n records in Phase-Two is

$$t_{\text{sort}} + x(t_{\text{bc}} + t_{\text{bcinbt}} + t_{\text{i/o}}) + t_{\text{iigat}} + t_{\text{iigddit}} + i(t_{\text{i/o}}) + 3t_{\text{i/o}}$$

where $i(t_{\text{i/o}})$ is writing each individual CDT to its respective backend and $3(t_{\text{i/o}})$ is writing the CINBT, IIGAT and IIGDDIT to the controller disk. The total time required to generate a large database using the two-pass approach is

$$n(t_{\text{load}} + t_{\text{get}} + t_{\text{process}} + t_{\text{ddit}} + t_{\text{cdt}}) + t_{\text{sort}} + x(t_{\text{bc}} + t_{\text{bcinbt}} + t_{\text{i/o}}) + t_{\text{iigat}} + t_{\text{iigddit}} + i(t_{\text{i/o}}) + 3(t_{\text{i/o}}) \text{ -- (2).}$$

We observe that there are some similar constants that appear in both equations (1) and (2) (i.e., $t_{\text{load}}$, $t_{\text{get}}$, $t_{\text{ddit}}$, $t_{\text{cdt}}$ and $t_{\text{process}}$). $T_{\text{get}}$, $t_{\text{load}}$, $t_{\text{cd}}$, and $t_{\text{ddit}}$ can be eliminated from both equations. They are calculated the same against the same number of records. Since they are constant factors that appear in both equations, removing them from each equation will not impact the overall comparative analysis. Tprocess also appears in both equations. This constant cannot be removed because in equation (1) it is linked to two I/O operations, whereas in equation (2) it is not. The resulting equations are:

$$n(t_{\text{process}} + t_{\text{iigat}} + t_{\text{iigddit}} + t_{\text{cinbt}} + 3t_{\text{i/o}} + t_{\text{write}}) \text{ --(1')}$$

$$n(t_{\text{process}}) + t_{\text{sort}} + x(t_{\text{bc}} + t_{\text{bcinbt}}) + t_{\text{I/o}}(x + i + 3) + t_{\text{ligat}} + t_{\text{ligaddit}} + t_{\text{sort}} \text{ -- } (2')$$

If we ignore the references to the meta data and concentrate solely on record processing and data loading we can simplify the equations to the following:

$$n(t_{\text{process}} + 2t_{\text{i/o}}\ t_{\text{write}}) \text{ -- } (1'')$$

$$nt_{\text{process}} + xt_{\text{bc}} + t_{\text{i/o}}(x + i + 3) + t_{\text{sort}} \text{ -- } (2'')$$

Now we see that the number of I/O requests is drastically reduced. The number of I/O requests in the new algorithm depends largely on the number of tracks that will be written to the backends instead of the number of records. If we view these two equations solely on the basis of loading the records, equations (1") and (2") will appear as

$$n(t_{\text{process}} + 2t_{\text{i/o}} + t_{\text{write}}) \text{ -- } (3)$$

$$nt_{\text{process}} + xt_{\text{i/o}} + t_{\text{sort}} \text{ -- } (4)$$

respectively. Notice from equation (4) we removed $t_{\text{i/o}}(i + 3)$. These I/O operations load the meta data onto disk. If the database is sufficiently large, then these I/O operations can be ignored.

These two operations are now in terms of record processing and I/O operations. We can see that deferring the I/O operations until Phase-Two reduces the number of I/O operations greatly. When generating large databases, equation (3) will perform on the order of (3n) while equation (4) performs on the order of (n + x). Since records are loaded in blocks, less I/O is required. This algorithm does prove to be advantageous over the current mode of operation.

## E. DETAILED DESIGN SPECIFICATION OF ALGORITHM

```
program Create_Reorg(input,output);
/*****************************************************************/
/*                                                                */
/*      This utility program can be used to generate a data      */
/*      over multiple backends/disks. A two pass approach        */
/*      is used in processing the data. Phase-One will scan      */
/*      the data, process the data, load this data to a tape     */
/*      and build three tables. Phase-Two will sort the tape     */
/*      produced in Phase-One, load the sorted data onto the     */
/*      disk, then load the tables built in Phase-One onto       */
/*      the meta data disk.                                      */
/*                                                                */
/*****************************************************************/


begin


Phase-One(...);
    if ((Good_Cluster_Distribution) and (Less_N%_Bad_Records)) then
Phase-Two(...)
    else
      Exit--Redefine Descriptor File;


end. /*  main  */
```

```
procedure Phase-One(...);


begin

 case TypeOperation of

   Create:
     begin
       get template file;
       get descriptor file;
       Build_AT(...);
       Build_DDIT(...);
     end;

   Reorg:
     begin
       get AT;
       get DDIT;
       remove Type C attributes from DDIT;
     end;

 end; /* case */

 repeat                       /* process group of records */
 load in records;
 get a record;

   repeat                     /* process each record */
   count record;
   scan record;
   Error_Check_Record(...);

   if good_record then        /* process good records */
     begin
       format record;
       Build_DDIT(...);
       Build_CDT(...);
       write record to work space;
     end;
```

```
    else                        /*  process bad records  */
      begin
        Bad_Record_Handler;
        count bad records;
      end;

    get record;
    until no_more_records

  write block of records to tape;
  until end_of_tape

  /*  CRITERIA: even distribution of records among clusters  */
  determine if there is a good cluster distribution;
  calculate percentage of bad records;

end;  /*  Phase-One  */
```

```
procedure Phase-Two(...);


begin

  tape sort by cluster id;

  repeat                          /*  load records to disk  */
  load records;
  get a record;

    repeat
    if (record_in_same_cluster) then
      begin
        if (recordsize + clustersize <= tracksize-2bytes) then
          begin                   /*  build a cluster  */
            add recordsize to clustersize;
            build cluster;
          end;

        else                      /* new cluster, same cluster id  */
          begin
            add recordsize to clustersize:
            determine backend to receive next cluster;
            load backend:
            Build_CDT(...);       /* for each individual backend  */
      end;

    else                          /*  new cluster, different cluster id */
      begin
        Build CINBT:
        add recordsize to clustersize;
        determine backend to receive next cluster;
        load backend:
        Build_CDT(...):
      end:
```

```
    get a record
    until no_more_records

  until end_of_tape

  /* load last cluster to backend */
  build CINBT;
  determine backend to receive next cluster;
  load backend;
  Build_CDT(...);

  /* load controller meta data */
  build IIGAT;
  build IIGDDIT;
  load IIGDDIT to controller;
  load IIGAT to controller;
  load CINBT to controller;

  /* load meta data to disk */

  case TypeOperation of

    Create:
      begin
        load AT to each backend;
        load DDIT to each backend;
        load individual CDT to appropriate backend;
      end;

    Reorg:
      begin
        load DDIT to each backend;
        load individual CDT to appropriate backend;

  end; /* case */

end; /* Phase-Two */
```

# IV. IMPLEMENTATION ISSUES

The user interface is the avenue in which the user accesses the database or interacts with the system. The user interface is usually characterized by the data model and its model-based data language. The data model allows the user to refer to the database in terms of its logical representation and the data language allows the user to write generic transactions and queries against the database. The user data model and language is always a high-level construct which is abstract enough so the user is not "bogged down" with the details of the database and the database system.

The kernel software of the of the multi-lingual, multi-model, multi-backend database system (MLDS, MMDS, MBDS) is the attribute-based data model (ABDM) and the attribute-based data language (ABDL). ABDL provides the user with a means of accessing and manipulating the database. ABDL provides the user with five primary operations and interactive dialogue. It is through these operations and dialogue that the user interacts with the system. It will be through the interactive dialogue that the algorithm will be incorporated into the system.

The interactive dialogue is menu driven. The menus are extensive, but organized simply. All menu items are organized as a multi-level hierarchy with top levels to indicate the type of dialogues and the low level to indicate the specific dialogue to follow.

The algorithm embodies many of the capabilities already existing in the system. Integration of the algorithm into MBDS should be done easily. The inputs needed to generate the database are the template, descriptors and record files. MBDS has the capability, through the user interface, to prompt the user for this information. The inputs needed to reorganize a database are the attribute table (AT) and the descriptor-to-descriptor-id table (DDIT). These tables are already in existence. As stated earlier, the interactive dialogue is menu driven. The algorithm will take advantage of the functions already developed. We propose to modify some of the existing menus of MBDS to integrate the algorithm into the system. The next few pages reflect how the menus should be changed and what options should be added to facilitate the incorporation of the algorithm. Upon entering MBDS the user will have the following menu appear on the screen

```
          The Multi-Lingual/Multi-Backend Database System

   Select an operation:

          (a)  -  Execute the attribute-based/ABDL interface
          (r)  -  Execute the relational/SQL interface
          (h)  -  Execute the hierarchical/DL/I interface
          (n)  -  Execute the network/CODASYL interface
          (f)  -  Execute the functional/DAPLEX interface
          (x)  -  Exit to the operation system.

   Select-> a
```

If the user is creating a new data base or reorganizing an existing database then he/she will select "a" as shown above. The next menu to appear will be

```
   The attribute-based/ABDL interface:

          (g)  -  Generate a database
          (o)  -  Reorganize an existing database
          (l)  -  Load a database
          (r)  -  Request interface
          (x)  -  Exit to operating system

   Select-> g
```

If the user is creating a new database, then he/she will select "g" as shown above. Next the user will be prompted for the number of backends:

```
          Enter number of backends->
```

The user will enter a numeric value such as "8". Next the system will provide the user with a series of prompts. These prompts are used to collect information about the template, descriptor and record files (meta data files). The user can enter data into the system two ways. He/she can have a file already built containing the meta data files or can generate the meta data files at this point. The following prompts will appear:

```
    What operation would you like to perform:
          (p)  -  Load template, descriptor and record files (predefined files)
          (g)  -  Load template, descriptor and record files (generate)
          (x)  -  Exit to operating system
          (z)  -  Exit and stop MBDS
```

If the user has already built the meta data files he/she will select "p" and the following prompts will appear, in succession, requesting the meta data file names

```
          Enter name of the file containing Template information:
          Enter name of the file containing Descriptor information:
          Enter name of the file containing Records to be loaded:
```

If the user does not have predefined meta data files then he/she will select "g" and the following series of prompts will appear:

```
    For building the Template file:
          Enter name of the file to be used to store template information:
          Enter database id:
          Enter number of templates for database-name:
          Enter number of attributes for template #1:
          Enter name of template #1:
          Enter attribute name #1 for template #1:
          Enter value type:
          Enter attribute name #2 for template #1:
          Enter value type:
          Enter number of attributes for template #2:
```

```
                    .  .
                    .  .

        Enter name of template #2:
        Enter attribute name #1 for template #2:
        Enter value type:
        Enter attribute name #2 for template #2:
        Enter value type:
        .

        .
        Enter number of attribute for template #n:
        Enter name of template #n:
        Enter attribute name #1 of template #n:
        Enter value type:

    For building the Descriptor file:
        Enter name of the file used to store descriptor information:
        Enter name of template file:
        Do you want attribute "attribute-name" to be a directory attribute:
        Enter the descriptor type for "attribute-name"(A,B,C):
        Enter upper bound for each descriptor in turn
            --Enter "@" to stop:
        Upper bound:
        Use "!" to indicate no lower bound exists
            --Enter "@" to stop:
        Lower bound:

    For building the record file:
        Enter name of the file containing records to be loaded:
```

The system now has the three files that it needs to actually begin using the algorithm. A message will appear on the screen informing the user, that all inputs are received and database generation has begun. The algorithm will proceed to process the records. If Phase-One is successful, the algorithm will proceed on to Phase-Two. If Phase-One fails, the system will send a message to the user. The message will read one of the following:

```
        PHASE-ONE FAILED!!! -- BAD RECORD DISTRIBUTION
        PHASE-ONE FAILED!!! -- DATA CORRUPT
```

The user will correct the problem. If Phase-One failed because of bad record distribution, the user should redefine his/her descriptor file. Specifically, the type-A range variables should be adjusted. If Phase-One failed because of corrupt data, then the user should check the file containing the bad data. This file was produced during the first pass. After the user has made the correction needed, he/she should re-start the process to generate the new database.

If the user's operation is to reorganize an existing database then at the menu shown below:

```
The attribute-based/ABDL interface:

        (g)   -  Generate a database
        (o)   -  Reorganize an existing database
        (l)   -  Load a database
        (r)   -  Request interface
        (x)   -  Exit to operating system

Select-> o
```

the user should select "o" as shown above. The user will receive a series of prompts requesting information about the database to be reorganized. The following prompt will appear

```
Enter number of backends:
Enter name of database to be reorganized:
Entering the name of the containing the records to be loaded:
```

Once the user enters the name of the database to be reorganized, the algorithm can now identify the tables needed as input. The system will now start actually using the algorithm. A message will appear on the screen informing the user that the reorganization of the specified database has begun. The algorithm will proceed to process the records. If Pass-One is successful, the algorithm will proceed on to Phase-Two. If pass-one fails, the system will send a message to the user. The message will read one of the following:

```
PHASE-ONE FAILED!!! -- BAD RECORD DISTRIBUTION
PHASE-ONE FAILED!!! -- DATA CORRUPT
```

The user will take whatever action is appropriate to eliminate the problem. Once the problem has been corrected the user will re-enter the system and restart the process.

# V. CONCLUSIONS

We have found that all database systems have  four major components: a data model and its model-based data language, a database, software and hardware. We have discussed these components in some detail over the past few chapters. The kernel system is the underlying sy3tem used to support the multi-backend database system (MBDS). The kernel system has its own data model and language known as the attribute-based data model and language (ABDM, ABDL). Like most database systems ABDL provides its users with the capability to access and manipulate its data. Unfortunately, as mentioned earlier, the kernel system does not lend itself to providing a fast and efficient method of generating large database. We know that through the kernel data language we can create these large databases with the use of the INSERT command. The INSERT command is a one-record-at-a-time operation. In order to store large databases onto disk we must use the one-record-at-a-time methodology. This methodology is not very efficient but it does work. We have learned that the DELETE command does not physically remove a record from the database but in reality, tags the record as longer active. These tagged records leave "holes" in the database creating fragmentation within the medium. If we collect the active records and then redistribute them over the backends we eliminate the problem of fragmentation. The entire thrust of this thesis is to provide a solution to the problem of database creation and garbage collection.

## A. A REVIEW OF THE RESEARCH

In this thesis, we have addressed the topic of database creation and/or reorganization (garbage collection) over multiple backends. Specifically, we have presented a methodology that will efficiently create very large databases of gigabytes on parallel computers and reorganize them when they have records that have been tagged for deletion. We have designed a utility program to by-pass the system's INSERT command, to load the data directly onto disk and create all necessary base data and meta data of the database.

In this thesis, we recognized two approaches (i.e., One-Pass and Two-Pass approaches) that may be taken with respect to database creation or garbage collection. We discussed the two methods and gave our reasons for selecting the Two-Pass approach as the best alternative.

The Two-Pass chosen methodology entailed two phases. The first phase is known as Phase-One. The first phase deals mostly with record processing. Also performed in the first pass is the initial building of the three directory tables. There is very little I/O processing performed during the first phase. The second phase is known as Phase-Two. The second phase is concerned primarily with loading data, both base and meta, onto the backends' disks. The loading of the data generates many I/O requests, so the second phase is I/O-bound. The separation of the data being processed and the data being loaded have reduced the number of I/O operations. Now records are loaded in clusters instead of individually.

We feel that the methodology presented in this thesis is sufficient for implementation. With the implementation an efficient means of generating large databases with even distributed over multiple backends will become a reality.

## B. SOME OBSERVATIONS AND INSIGHT

The multi-lingual, multi-model, multi-backend database system (MLDS,MMDS, MBDS) is a very powerful, yet simple system. The power of the system lies in its ability to provide one user access to a neighboring database which was created under a different database model, using his/her own data language. The attribute-based data model and language is the kernel system which supports the MLDS, MMDS and MBDS. There are considerable design, development and testing efforts in making this system a research vehicle for new research undertakings. This thesis, once implemented, will provide an additional capability to an already powerful system. Some areas of future research in database creation, would be (1) expanding this algorithm to generate more than one database at a time, and (2) placing a new database on the backends that already have resident databases.

# APPENDIX A - ATTRIBUTE TABLE PROGRAM SPECIFICATIONS

```
#include <stdio.h>
#include "flags.def"
#include "dblocal.def"
#include "beno.def"
#include "commdata.def"
#include "dirman.def"
#include "tmpl.def"
#include "dirman.ext"


struct  ddit_definition  *ATM_FIND(attribute, desc_type, AT)
   /* Find an attribute in AT and return the pointer to its DDIT.  Set */
   /* desc_type to the type of descriptors defined on the attribute. */
   char   attribute[],
          *desc_type; /* not C, C, AT_NOTFOUND or AT_DELETED */
   struct at_tbl_definition *AT; /* attribute table */
{
   int pos; /* position of attribute in Attribute Table */

#ifdef EnExFlagg
   printf("Enter ATM_FIND\n");
   fflush(stdout);
#endif

   /* find attribute in AT */
   pos = AT_binsearch(AT, attribute);

   /* if attribute not in AT */
   if (pos == -1 || AT->at_entry[pos].at_desc_type == AT_DELETED) {
      *desc_type = AT_NOTFOUND;

#ifdef EnExFlagg
      printf("Exit1 ATM_FIND\n\n");
      fflush(stdout);
#endif
```

```
      return(NULL);
   } else {
      *desc_type = AT->at_entry[pos].at_desc_type;

#ifdef EnExFlagg
      printf("Exit2 ATM_FIND\n\n");
      fflush(stdout);
#endif

      return(AT->at_entry[pos].at_ddit_ptr);
   }

} /* end ATM_FIND */



ATM_INSERT(attr_name, attr_id, desc_type, ddit_ptr, AT)
   /* Insert an attribute into AT. */
   char    attr_name[];
   int     attr_id;
   char    desc_type; /* not C, C, AT_NOTFOUND or AT_DELETED */
   struct  ddit_definition  *ddit_ptr; /* pointer to first DDIT element */
   struct  at_tbl_definition  *AT; /* attribute table */
{
   int position = 0, /* index to AT for attribute */
       compare, /* return value from string comparison of attributes */
       i;

#ifdef EnExFlagg
   printf("Enter ATM_INSERT\n");
   fflush(stdout);
#endif

   /* the attribute table is maintained sorted by attribute name; if not the */
   /* first entry the table must be checked for fullness; if the table is */
   /* full, attributes marked for deletion are removed; if the table is still */
   /* full (no attributes were marked for deletion) an error condition exists */

   if (!AT->at_no_entry) {
      /* if first entry into table simply insert */
      strcpy(AT->at_entry[0].at_AttrName, attr_name);
      AT->at_entry[0].at_AttrId = attr_id;
      AT->at_entry[0].at_desc_type = desc_type;
      AT->at_entry[0].at_ddit_ptr = ddit_ptr;
      ++AT->at_no_entry;
```

```
    } else {
        /* if full table remove deleted entries */
        if (AT->at_no_entry == AT_MAX_ENTRIES)
            AT_remove_del(AT);
        /* if table still full */
        if (AT->at_no_entry == AT_MAX_ENTRIES) {
            printf("ERROR: AT is full in ATM_INSERT().\n");
            sleep(ErrDelay);
        } else {
            /* find correct position */
            while ((position < AT->at_no_entry) &&
                (compare = strcmp(attr_name,AT->at_entry[position].at_AttrName))>0)
                ++position;

            /* check for duplicate entries */
            if (!compare && (AT->at_entry[position].at_desc_type != AT_DELETED))
              printf("ATM_INSERT     attribute is already in AT\n");
            else {
                /* shift down table entries */
                for (i = AT->at_no_entry - 1; i >= position; --i) {
                  strcpy(AT->at_entry[i+1].at_AttrName,AT->at_entry[i].at_AttrName);
                  AT->at_entry[i + 1].at_AttrId = AT->at_entry[i].at_AttrId;
                  AT->at_entry[i + 1].at_desc_type = AT->at_entry[i].at_desc_type;
                  AT->at_entry[i + 1].at_ddit_ptr  = AT->at_entry[i].at_ddit_ptr;
                }
                /* insert new entry */
                strcpy(AT->at_entry[position].at_AttrName, attr_name);
                AT->at_entry[position].at_AttrId = attr_id;
                AT->at_entry[position].at_desc_type = desc_type;
                AT->at_entry[position].at_ddit_ptr  = ddit_ptr;
                ++AT->at_no_entry;
            }
        }
    }

#ifdef EnExFlagg
    printf("Exit ATM_INSERT\n\n");
    fflush(stdout);
#endif

    return (position);

} /* end ATM_INSERT */
```

```
static ATM_DELETE(attribute,AT)
   /* Mark an attribute in AT for deletion. */
   char attribute[];
   struct at_tbl_definition *AT; /* attribute table */
{
   int  position; /* position of attribute in AT table */

   /* find attribute in AT */
   position = AT_binsearch(AT,attribute);

   if (position != -1)
      /* mark the attribute for deletion */
      AT->at_entry[position].at_desc_type = AT_DELETED;
   else
      SysError(5, "ATM_DELETE");

} /* end ATM_DELETE */



ATM_UPDATE(attribute, new_ddit_ptr, AT)
   /* Update the dditptr for an attribute in AT. */
   char    attribute[];
   struct ddit_definition  *new_ddit_ptr; /* ptr to first element of DDIT */
   struct at_tbl_definition  *AT; /* attribute table */
{
   int position; /* index to AT */

#ifdef EnExFlagg
   printf("Enter ATM_UPDATE\n");
   fflush(stdout);
#endif

   /* find attribute in AT */
   position = AT_binsearch(AT, attribute);
   if (position != -1)
      /* update ddit ptr in AT */
      AT->at_entry[position].at_ddit_ptr = new_ddit_ptr;
   else {
      SysError(5, "ATM_UPDATE");
      sleep(ErrDelay);
   }
#ifdef EnExFlagg
   printf("Exit ATM_UPDATE\n\n");
   fflush(stdout);
#endif
} /* end ATM_UPDATE */



struct  at_tbl_definition  *AT_lookuptbl(dbid)
   /* Find the AT for a database and return a pointer to it */
   char dbid[];
{
   struct db_info *db_ptr, *DB_find();

#ifdef EnExFlagg
   printf("Enter AT_lookuptbl\n");
   fflush(stdout);
#endif

   /* find AT for database dbid */
   db_ptr = DB_find(dbid);

   if (db_ptr) {
      /* AT is found */

#ifdef EnExFlagg
      printf("Exit1 AT_lookuptbl\n\n");
      fflush(stdout);
#endif

      return(db_ptr->db_at_pointer);
   } else {
      /* AT not found */

#ifdef EnExFlagg
      printf("Exit2 AT_lookuptbl\n\n");
      fflush(stdout);
#endif

      return(NULL);
   }

} /* end AT_lookuptbl */
```

```
AT_binsearch(AT, attribute)
   /* Find an attribute in AT using binary search and return its position */
   struct  at_tbl_definition  *AT;  /* Attribute Table */
   char     attribute[];
{
   int      high,     /* highest index in portion of tbl being searched */
            low = 0, /* lowest index in portion of tbl being searched */
            mid,      /* index who's attr name is being compared */
            compare; /* return value from string comparison of attributes */

#ifdef EnExFlagg
   printf("Enter AT_binsearch\n");
   fflush(stdout);
#endif

   high = AT->at_no_entry - 1;

   while (low <= high) {
      mid = (low + high) / 2;
      compare = strcmp(attribute, AT->at_entry[mid].at_AttrName);
      if (!compare) {

#ifdef EnExFlagg
      printf("Exit1 AT_binsearch\n\n");
      fflush(stdout);
#endif
         return(mid);
      } else
         if (compare < 0)
            high = mid - 1;
         else
            low = mid + 1;

   } /* end while */

   /* attribute not found in AT */

#ifdef EnExFlagg
   printf("Exit2 AT_binsearch\n\n");
   fflush(stdout);
#endif
   return (-1);

} /* end AT_binsearch */
```

```
static AT_remove_del(AT)
   /* Remove attributes marked for deletion from AT. */
   struct  at_tbl_definition  *AT; /* attribute table */
{
   int i, j = 0;     /* indexes to AT table */

   for (i = 0; i < AT->at_no_entry; ++i)
      if (AT->at_entry[i].at_desc_type != AT_DELETED) {
         /* keep this attribute */
         strcpy(AT->at_entry[j].at_AttrName, AT->at_entry[i].at_AttrName);
         AT->at_entry[j].at_AttrId = AT->at_entry[i].at_AttrId;
         AT->at_entry[j].at_desc_type = AT->at_entry[i].at_desc_type;
         AT->at_entry[j].at_ddit_ptr  = AT->at_entry[i].at_ddit_ptr;
         ++j;
      }

   /* update number of entries in AT */
   AT->at_no_entry = j;
   AT->at_write_required = TRUE;

} /* end AT_remove_del */
```

# APPENDIX B - DESCRIPTOR TABLE PROGRAM SPECS.

```
#include <stdio.h>
#include "flags.def"
#include "dblocal.def"
#include "beno.def"
#include "commdata.def"
#include "dirman.def"
#include "dirman.ext"


struct ddit_definition *
DM_INSERT_DDIT(descriptor, val_type, ddit_list_header, new_desc_id)
   /* Add a descriptor to DDIT. If the descriptor is added to the beginning, */
   /*  return a pointer to it. */
   struct desc_definition  *descriptor;
   char   val_type;
   struct ddit_definition  *ddit_list_header;/* first element in DDIT list */
   struct DescId  *new_desc_id;
{
   int     compare;
   struct ddit_definition *create_ddit_node(), *new_ddit, *next_ddit,*prev_ddit;

#ifdef EnExFlag
   printf("Enter DM_INSERT_DDIT\n");
#endif

   new_ddit = create_ddit_node();

#ifdef m_pr_flag
   printf("new_ddit = %o\n", new_ddit);
#endif

   /* place descriptor into ddit element */
   strcpy(new_ddit->lower, descriptor->lower);
   strcpy(new_ddit->upper, descriptor->upper);
   di_cpy(&(new_ddit->ddit_did), new_desc_id);

   if (!ddit_list_header) {
      /* creating new list */
```

```
#ifdef EnExFlag
    printf("Exit1 DM_INSERT_DDIT\n");
#endif

    return(new_ddit);
  } else {
    /*find proper place in existing list */
    prev_ddit = NULL;
    next_ddit = ddit_list_header;
    if (next_ddit->lower[0] == NOBOUND && next_ddit->upper[0] ==
NOBOUND) {
      /* the first one is catchall descriptor; skip it */
      prev_ddit = next_ddit;
      next_ddit = next_ddit->next_ddit_definition;
    }
    while (next_ddit) {
      compare = datacmp(next_ddit->upper, new_ddit->upper, val_type);
      if (compare < 0) {
        prev_ddit = next_ddit;
        next_ddit = next_ddit->next_ddit_definition;
      } else
        break;
    }

    /* if new ddit is at beginning of list */
    if (!prev_ddit) {
      new_ddit->next_ddit_definition = ddit_list_header;

#ifdef EnExFlag
      printf("Exit2 DM_INSERT_DDIT\n");
#endif

      return(new_ddit);
    } else {
      /* new ddit somewhere after first element */
      new_ddit->next_ddit_definition = prev_ddit->next_ddit_definition;
      prev_ddit->next_ddit_definition = new_ddit;
#ifdef EnExFlag
            printf("Exit3 DM_INSERT_DDIT\n");
#endif
            return(NULL);
        }

    } /* end if (ddit_list_header) */

} /* end DM_INSERT_DDIT */
```

## APPENDIX C - CLUSTER DEFINITION TABLE PROGRAM SPECS.

```
#include <stdio.h>
#include "flags.def"
#include "dblocal.def"
#include "beno.def"
#include "commdata.def"
#include "dirman.def"
#include "dirman.ext"


struct  cdt_definition   *CDTM_INSERT(DT, d_i_s, cid)
   /* Add an entry to cluster-definition table and return a pointer to it.
    * It will also update DT if necessary (this happens if one or more of the
    * descriptor ids defining the cluster are not in DT). Update DTCT.  */
   struct dt_definition   *DT;  /* descriptor table */
   struct des_id_set   *d_i_s;
   int      cid;
{
   struct cdt_definition *create_cdt_node(), *new_cdt_ptr;
   struct did_link_definition   *create_did_link_node(), *desc_ptr,
                                 *prev_desc_ptr;
   struct dtct_definition   *create_dtct_node(), *new_dtct_ptr;
   int      k, ind;

#ifdef EnExFlag
   printf("Enter CDTM_INSERT\n");
#endif

   /* sort the descriptor-id set */
   DescSort(d_i_s);

   /* allocate new CDT entry */
   new_cdt_ptr = create_cdt_node();

#ifdef m_pr_flag
   printf("new_cdt_ptr = %o\n", new_cdt_ptr);
#endif

   /* store the information in the new entry */
   new_cdt_ptr->cdt_clus_no = cid;
   /* copy the descriptor ids into the cdt entry */
   new_cdt_ptr->cdt_no_desc = d_i_s->dis_desc_count;
   desc_ptr = create_did_link_node();

#ifdef m_pr_flag
   printf("desc_ptr = %o\n", desc_ptr);
#endif

   di_cpy(&(desc_ptr->dld_did), &(d_i_s->dis_dids[0]));
   new_cdt_ptr->cdt_did_pointer = desc_ptr;
   for (k = 1; k < d_i_s->dis_desc_count; ++k) {
      prev_desc_ptr = desc_ptr;
      desc_ptr = create_did_link_node();

#ifdef  m_pr_flag
      printf("desc_ptr = %o\n", desc_ptr);
#endif
      di_cpy(&(desc_ptr->dld_did), &(d_i_s->dis_dids[k]));
      prev_desc_ptr->next_did_link_definition = desc_ptr;

   } /* end for */

   desc_ptr->next_did_link_definition = NULL;

   /* update DT and DTCT */
   for (k = 0; k < d_i_s->dis_desc_count; ++k) {
      /* find the descriptor id in DT */
      if ((ind = binsr_dt(&(d_i_s->dis_dids[k]), DT)) == -1)
         /* the descriptor id is not in DT; add it to DT */
         ind = CDTM_DT_INSERT(&(d_i_s->dis_dids[k]), DT);
      new_dtct_ptr = create_dtct_node();

#ifdef  m_pr_flag
      printf("new_dtct_ptr = %o\n", new_dtct_ptr);
#endif
      new_dtct_ptr->dtct_cdt_pointer = new_cdt_ptr;
      new_dtct_ptr->next_dtct_definition = DT->dt_entry[ind].dt_dtct_pointer;
      DT->dt_entry[ind].dt_dtct_pointer = new_dtct_ptr;
      DT->dt_entry[ind].dt_clus_count++;

   }/* end for */
   DT->dt_write_required = TRUE;
```

```
#ifdef EnExFlag
   printf("Exit CDTM_INSERT\n\n");
#endif

   return (new_cdt_ptr);

}/* end CDTM_INSERT */



CDTM_DT_INSERT(d_id, DT)
   /* Add a descriptor id to descriptor table (DT) if it not already there.
    * Return the position of the descriptor id in DT. */
   struct  DescId  *d_id;
   struct  dt_definition  *DT;  /* descriptor table */
{
   int       pos, k;

#ifdef EnExFlag
   printf("Enter CDTM_DT_INSERT\n");
#endif

   /* check to see if the descriptor id is already in DT ... done where called
   if ((pos = binsr_dt(d_id, DT)) != -1) {
      #* descriptor id is already in DT *#
#ifdef EnExFlag
      printf("Exit1 CDTM_DT_INSERT\n\n");
#endif
      return (pos);
   } */

   /* the descriptor id is not in DT: add it */

#ifdef special_flag
  printf("DT->dt_no_did = %d\n", DT->dt_nc_did);
#endif

   /* Check for room */
   if (DT->dt_no_did  >= DT_MAX_DESC) {
      SysError(9, "CDTM_DT_INSERT");
      sleep(ErrDelay);
   }
```

```
    /* find the appropriate position in DT for the descriptor id (DT is ranked
     * in ascending order by descriptor ids) */
    for (pos = 0; pos < DT->dt_no_did &&
            strcmp(DT->dt_entry[pos].dt_did.did, d_id->did) < 0; ++pos)
       ;

    /* the new descriptor id should be added to DT at position 'pos' */
    /* move down the descriptor ids in DT to make space for the new one */
    for (k = DT->dt_no_did; k >= pos; --k) {
       di_cpy(&(DT->dt_entry[k].dt_did), &(DT->dt_entry[k-1].dt_did));
       DT->dt_entry[k].dt_clus_count  = DT->dt_entry[k-1].dt_clus_count;
       DT->dt_entry[k].dt_dtct_pointer = DT->dt_entry[k-1].dt_dtct_pointer;
    }

    /* add the new descriptor id */
    di_cpy(&(DT->dt_entry[pos].dt_did), d_id);
    DT->dt_entry[pos].dt_clus_count = 0;
    DT->dt_entry[pos].dt_dtct_pointer = NULL;

    /* increment the number of descriptor ids in DT */
    ++DT->dt_no_did;

#ifdef EnExFlag
    printf("Exit2 CDTM_DT_INSERT\n\n");
#endif

    return (pos);

} /* end CDTM_DT_INSERT */
```

# APPENDIX D - RECORD CHECKER PROGRAM SPECS.

```
#include   "flags.def"
#include   <stdio.h>
#include   "beno.def"
#include   "commdata.def"
#include   "reqprep.def"
#include   "reqprep.ext"


chk_ParsedTrafUnit(ParsedRequestsPtr, dbid, err_msg)
    /* Check all the requests in a traffic unit against the record template */
    struct  req_index_definition   *ParsedRequestsPtr;
    char     err_msg[], dbid[];
{
    int      k, tmpl_index;
    struct  REQtbl_definition *req_ptr;
    struct  rtemp_definition   *tmpl_ptr,
                              *get_tmpl_ptr();

#ifdef pr_flag
int z=0;
#endif
#ifdef  EnExFlag
    printf("Enter chk_ParsedTrafUnit \n");
#endif

    for (k = 0; (req_ptr = ParsedRequestsPtr->req_tbl_pointer[k]) != NULL; ++k)
     {
#ifdef pr_flag
      while(req_ptr->req_tbl[z][0]!=EORequest)
           printf("%s\n",req_ptr->req_tbl[z++]);
      printf("%s\n",req_ptr->req_tbl[z+1]);
#endif
     /* Get the record template. Template name is found as follows:
                    req_tbl[4][0] = request type
                  req_tbl[tmpl_index] = template name pointer */
    if((req_ptr->req_tbl[4][0]-'0')==INSERT)
      tmpl_index=7;
    else
      tmpl_index=8;
        /*
        switch (req_ptr->req_tbl[4][0] - '0') {
            case INSERT:
                 tmpl_index = 7;
                 break;
            case RETRIEVE:
                 case RET_COM:
            case DELETE:
            case UPDATE:
                 tmpl_index = 8;
        }
        */

        tmpl_ptr = get_tmpl_ptr(dbid,req_ptr->req_tbl[tmpl_index]);

        if (!chk_request(ParsedRequestsPtr->req_tbl_pointer[k],tmpl_ptr,err_msg))
          {
#ifdef  EnExFlag
          printf("Exit1 chk_ParsedTrafUnit \n");
#endif
          return(FALSE);
          }
      } /* end of for loop */

    /* all the parsed requests are ok */
#ifdef  EnExFlag
    printf("Exit2 chk_ParsedTrafUnit \n");
#endif
    return(TRUE);

}/* end chk_ParsedTrafUnit */
```

```
chk_request(req_ptr, rtemp_ptr, err_msg)
  /* Purpose:                                                          */
  /*    This procedure checks the validity of a request by comparing it   */
  /* againest recored template.  It returns TRUE if the request is valid;  */
  /* otherwise, it returns FALSE and an error message.     */

  struct  REQtbl_definition  *req_ptr;
  struct  rtemp_definition  *rtemp_ptr;
  char     err_msg[];
{
  char     flag_attr;
  int      flag_insrt, flag_dlt, flag_rtrv, flag_updt;
  int      j,
           mod_type;
  char     chk_attr_name();


#ifdef  EnExFlag
  printf("Enter chk_requesr \n");
#endif

  /* check request type */
  switch (req_ptr->req_tbl[4][0] - '0')
   {
    case INSERT:
          flag_insrt = chk_insrt_rec(req_ptr,rtemp_ptr,err_msg);
#ifdef  EnExFlag
          printf("Exit1 chk_request \n");
#endif
          return(flag_insrt);
          break;

    case DELETE:
          j = 6;
          flag_dlt = chk_non_insrt_q (req_ptr,&j,rtemp_ptr,err_msg);
#ifdef  EnExFlag
          printf("Exit2 chk_request \n");
#endif
          return(flag_dlt);
          break;
```

```
      case RET_COM:
      case RETRIEVE:
           j = 6;
           flag_rtrv = chk_non_insrt_q (req_ptr,&j,rtemp_ptr,err_msg);
           if (!flag_rtrv)
             {
#ifdef  EnExFlag
              printf("Exit3 chk_request \n");
#endif
              return (FALSE);
             }
           j++; /* skip EOQuery */
           /* check target list */
           while ( req_ptr->req_tbl[j][0] != ETList )
             {
              flag_attr = chk_attr_name(rtemp_ptr,req_ptr->req_tbl[j]);
              if ( flag_attr == '0' )
                {
                 concat("invalid attribute name: ",req_ptr->req_tbl[j],err_msg);
#ifdef  EnExFlag
                 printf("Exit4 chk_request \n");
#endif
                 return(FALSE);
                }
              j++; /* skip attribute name */
              j++; /* skip the aggregate */
             }/* end while ( req_ptr->req_tbl[j][0] != ETList ) */
           j++;
           /* check the attribute for by clause */
           if ( strcmp(req_ptr->req_tbl[j],"000") != 0 )
             {
              flag_attr = chk_attr_name(rtemp_ptr,req_ptr->req_tbl[j]);
              if ( flag_attr == '0' )
                {
#ifdef  EnExFlag
                 printf("Exit5 chk_request \n");
#endif
                 concat("invalid attribute name : ",req_ptr->req_tbl[j],err_msg);
                 return(FALSE);
                }
             }
#ifdef  EnExFlag
           printf("Exit6 chk_request \n");
#endif
```

```
          return(TRUE);
          break;

      case UPDATE:
          j = 6;
          flag_updt = chk_non_insrt_q(req_ptr,&j,rtemp_ptr,err_msg);
          if (!flag_updt)
            {
#ifdef  EnExFlag
             printf("Exit7 chk_request \n");
#endif
             return(FALSE);
            }
          j++; /* skip EOQuery */
          mod_type = req_ptr->req_tbl[j][0] - '0';
          j++; /* skip modifier type */
          /* check the attr-being-modified */
          flag_attr = chk_attr_name(rtemp_ptr,req_ptr->req_tbl[j]);
          if ( flag_attr == '0' )
            {
#ifdef  EnExFlag
             printf("Exit8 chk_request \n");
#endif
             concat("invalid attribute name being modified: ",
                                        req_ptr->req_tbl[j], err_msg);
             return(FALSE);
            }
          j++; /* skip attribute being modified */
          /* check modifier type */
          switch ( mod_type )
            {
             case MT0:
                  /* check the new value for valid value type */
                  /* <<< TBC >>> */
                  break;

             case MT1:
             case MT2:
                  flag_attr = chk_attr_name(rtemp_ptr,req_ptr->req_tbl[j]);
                  if ( flag_attr == '0' )
                    {
#ifdef  EnExFlag
                     printf("Exit9 chk_request \n");
#endif
```

```
                    concat("invalid base attribute name: ",
                                   req_ptr->req_tbl[j], err_msg);
                    return(FALSE);
                   }
                  break;

              case MT3:
                  flag_attr = chk_attr_name(rtemp_ptr,req_ptr->req_tbl[j]);
                  if (flag_attr == '0') {
                    concat("invalid base attribute name: ",
                                  req_ptr->req_tbl[j], err_msg);
#ifdef  EnExFlag
                    printf("Exit10 chk_request \n");
#endif
                    return(FALSE);
                  }
                  j++; /* skip base attribute */
                  while ( req_ptr->req_tbl[j][0] != EOExpr )
                  j++;
                  j += 2; /* skip EOExpr and number of predicates */
                  flag_attr = chk_non_insrt_q(req_ptr,&j,rtemp_ptr,err_msg);
                  if (!flag_attr) {
#ifdef  EnExFlag
                     printf("Exit11 chk_request \n");
#endif
                     return (FALSE);
                  }
                  break;

              case MT4:
                  flag_attr = chk_attr_name(rtemp_ptr,req_ptr->req_tbl[j]);
                  if ( flag_attr == '0' )
                    {
                    concat("invalid base attribute name: ",
                                    req_ptr->req_tbl[j],err_msg);
#ifdef  EnExFlag
                    printf("Exit12 chk_request \n");
#endif
                    return(FALSE);
                   }
                  break;

              default:
                  concat("invalid modifier type",(mod_type + '0'),err_msg);
```

```
#ifdef  EnExFlag
                printf("Exit13 chk_request \n");
#endif
                return(FALSE);
                break;
           }/* end switch ( mod_type ) */

#ifdef  EnExFlag
         printf("Exit14 chk_request \n");
#endif
         return(TRUE);
         break;

    default:
         concat("invalid request type: ",req_ptr->req_tbl[4],err_msg);
#ifdef  EnExFlag
         printf("Exit15 chk_request \n");
#endif
         return(FALSE);
         break;
   }/* end switch ( req_ptr->req_tbl[4][0] - '0' ) */

#ifdef  EnExFlag
  printf("Exit chk_request \n");
#endif

}/* end chk_request */


chk_insrt_rec(req_ptr, rtemp_ptr, err_msg)
   /* Purpose:                                                  */
   /*    This routine checks if all (and only) attribute names in   */
   /* record template are in an insert request.  It also checks the */
   /* value type associated with each attribute.  It returns TRUE   */
   /* if ok; otherwise it returns FALSE and an error message.         */

   struct  REQtbl_definition  *req_ptr;
   struct  rtemp_definition  *rtemp_ptr;
   char     err_msg[];
{
   int      flag_ins;
   int      k;
   int      no_kwrd;
   char     str_no_attr[5];
```

```
#ifdef  EnExFlag
   printf("Enter chk_insrt_rec \n");
#endif

   no_kwrd = str_to_num(req_ptr->req_tbl[5]);
   if ( no_kwrd != rtemp_ptr->no_entries )
     {
      num_to_str(rtemp_ptr->no_entries, str_no_attr);
      concat("number of keywords in the insert request should be: ",
                                                   str_no_attr, err_msg);
#ifdef  EnExFlag
      printf("Exit1 chk_insrt_rec \n");
#endif
      return(FALSE);
     }
   for ( k = 0;    k < rtemp_ptr->no_entries;    ++k )
     {
      flag_ins = insrt_attr_name(rtemp_ptr->rt_entry[k].attr_name,
                                 rtemp_ptr->rt_entry[k].value_data_type,
                                 req_ptr,err_msg);
      if (!flag_ins)
        {
#ifdef  EnExFlag
         printf("Exit2 chk_insrt_rec \n");
#endif
         return(FALSE);
        }
     }/* end for */
#ifdef  EnExFlag
   printf("Exit3 chk_insrt_rec \n");
#endif
   return(TRUE);

}/* end chk_insrt_rec */


insrt_attr_name(att_name, att_val_type, req_ptr, err_msg)
   /* Purpose:                                                  */
   /*    This routine checks if attribute name in record template is */
   /* in the insert request, and also checks the validity of the     */
   /* value type associated with attribute name.  It returns TRUE if */
   /* attribute name exsists and valid valuetype; otherwise, it      */
   /* returns FLASE                                               */

   struct  REQtbl_definition  *req_ptr;
   char     att_name[];
```

```
    char    att_val_type;
    char    err_msg[];

 {
    int     j;
    int     flag_value;


#ifdef  EnExFlag
    printf("Enter insrt_attr_name \n");
#endif
    j = 6;
    while ( req_ptr->req_tbl[j][0] != EORecord )
      {
       if ( strcmp(att_name,req_ptr->req_tbl[j]) == 0 )
         {
          j++;
          flag_value = chk_value_type(att_val_type,req_ptr->req_tbl[j]);
          if (!flag_value)
            {
             concat("invalid value type for attribute: ",
                                          req_ptr->req_tbl[j-1], err_msg);
#ifdef  EnExFlag
             printf("Exit1 chk_insrt_rec \n");
#endif
             return(FALSE);
            }
#ifdef  EnExFlag
          printf("Exit2 chk_insrt_rec \n");
#endif
          return(TRUE);
         }
       j = j + 2; /* skip attribute name and attribute value */

      }/* end while */

    concat("missing attribute name: ", att_name, err_msg);
#ifdef  EnExFlag
    printf("Exit3 chk_insrt_rec proc \n");
#endif
    return(FALSE);

 }/* end insrt_attr_name */
```

```
chk_non_insrt_q(req_ptr, i, rtemp_ptr, err_msg)
   /* Purpose:                                                      */
   /*    This procedure checks the validity of the query part of    */
   /* non-insert request.  It returns TRUE if valid; otherwise, it */
   /* returns FALSE and an error message.                           */

   struct  REQtbl_definition  *req_ptr;
   struct  rtemp_definition  *rtemp_ptr;
   char    err_msg[];
   int     *i;
{
   int     flag_value;
   char    chk_attr_name ( );
   char    flag_attr;


#ifdef  EnExFlag
   printf("Enter chk_non_insrt_q \n");
#endif
   while ( req_ptr->req_tbl[*i][0] != EOQuery )
     {
      flag_attr = chk_attr_name(rtemp_ptr,req_ptr->req_tbl[*i]);
      if ( flag_attr == '0' )
        {
         concat("invalid attribute name: ", req_ptr->req_tbl[*i], err_msg);
#ifdef  EnExFlag
         printf("Exit1 chk_non_insrt_q \n");
#endif
         return (FALSE);
        }
      ++*i; /* skip attribute name */
      ++*i; /* skip relational operator */
      flag_value = chk_value_type(flag_attr,req_ptr->req_tbl[*i]);
     if (!flag_value)
        {
         concat("invalid value for attribute: ",
                                    req_ptr->req_tbl[*i-2], err_msg);
#ifdef  EnExFlag
         printf("Exit2 chk_non_insrt_q \n");
#endif
         return(FALSE);
        }
        ++*i; /* skip attribute value */
        if ( req_ptr->req_tbl[*i][0] == EOConj )
           ++*i; /* skip EOConj */

      }/* end while */

#ifdef  EnExFlag
   printf("Exit3 chk_non_insrt_q \n");
#endif
   return(TRUE);

}/* end chk_non_insrt_q */


char  chk_attr_name(rtemp_ptr, attribute)
   /* Purpose:                                                        */
   /*    This procedure checks if an attribute name exsists in the    */
   /* record template.  If the attribute is in the record template,   */
   /* it returns the type of value (i, s, f,...) for the attribute.   */
   /* If the attribute is not in the record template, it returns '0'. */

   struct  rtemp_definition  *rtemp_ptr;
   char      attribute[];
{
   int       i;


#ifdef  EnExFlag
   printf("Enter chk_attr_name \n");
#endif
   for ( i = 0;   i < rtemp_ptr->no_entries;   ++i )
     if ( strcmp(rtemp_ptr->rt_entry[i].attr_name,attribute) == 0 )
        {
#ifdef  EnExFlag
        printf("Exit1 chk_attr_name \n");
#endif
        return(rtemp_ptr->rt_entry[i].value_data_type);
        }

#ifdef  EnExFlag
   printf("Exit2 chk_attr_name \n");
#endif
   return('0');

}/* end chk_attr_name */
```

```
chk_value_type(value_type, val)
   /* Purpose:                                                    */
   /*    This procedure checks the validity of a value.  It returns */
   /* TRUE if valid; otherwise, it returns FALSE.                   */

   char     value_type;
   char     val[];
{
   int      i,negflag=0;

#ifdef  EnExFlag
   printf("Enter chk_value_type \n");
#endif

   switch ( value_type )
     {
      case 'i':
      /* allow for negative numbers */
         if((val[0] < '0' || val[0] > '9')&&(val[0]!='-'))
           return(FALSE);
       else
           if(val[0]==' ')
           /* since it is negative, make sure numbers follow,
            * not just '-' */
                negflag=1;

         for(i=1; val[i]!='\0';    ++i )
           if(val[i] < '0' || val[i] > '9') {
#ifdef  EnExFlag
              printf("Exit1 chk_value_type \n");
#endif
              return(FALSE);
             }

         /* scold the user if just a negative sign */
         if((negflag)&&(i==1))
           return(FALSE);

#ifdef  EnExFlag
         printf("Exit2 chk_value_type \n");
#endif
         return(TRUE);
         break;
```

```
        case 's':
#ifdef  EnExFlag
            printf("Exit3 chk_value_type \n");
#endif
            return(TRUE);
            break;

        case 'f':
#ifdef  EnExFlag
            printf("Exit4 chk_value_type \n");
#endif
            return(TRUE);
            break;

        default :
#ifdef  EnExFlag
            printf("Exit5 chk_value_type \n");
#endif
            return(FALSE);
            break;

      }/* end switch */

#ifdef  EnExFlag
   printf("Exit6 chk_value_type \n");
#endif

}/* end chk_value_type */
```

# LIST OF REFERENCES

1. Korth, H. F. and Siberschatz, K., *Database System Concepts*, p.348, McGraw- Hill Book Co., 1986

2. MacLennan, B. J., *Principles of Programming Languages: Design, Evaluation and Implementation*, 2d ed., p.432, CBS College Publishing, 1987.

3. Stubbs, D. F., and Webre, N. W., *Data Structures with Abstract Data Types and Modula-2*, p. 396, Brooks/Cole Publishing Co., 1987.

4. Demurjian, S. A. and Hsiao, D. K., "New Directions in Database-Systems Research and Development," Technical Report, NPS-52-85-001, Naval Postgraduate School, Monterey, California, February 1985.

5. Banerjee, J. and Hsiao, D. K., "The Use of a Database Machine for Supporting Relational Databases," *Proceedings 5th Workshop on Computer Architecture for Nonnumeric Processing* (August 1987).

6. Benerjee, J., Hsiao, D. K., and Ng, F., "Database Transformation, Query Transformations and Performance Analysis of a Database Computer in Supporting Hierarchical Database Management," *IEEE Transactions on Software Engineering Vol. SE-6, No. 1*, (January 1980).

7. Benerjee, J. and Hsiao, D. K., "A Methodology for Supporting Existing CODASYL Databases with New Database Machines," *Proceedings of National ACM Conference* (1978).

8. Macy, G., "Design and Analysis of an SQL Interface for a Multi-Backend Database System," M. S. Thesis, Naval Postgraduate School, Monterey, California, March 1984.

9. Weisher, D., "Design and Analysis of a Complete Hierarchical Interface for a Multi-Backend Database System," M. S. Thesis, Naval Postgraduate School, Monterey, California, June 1984.

10. Wortherly, C. R., "Design and Analysis of a Network Interface for a Multi-Backend Database System," M. S. Thesis, Naval Postgraduate School, Monterey, California, December 1985.

11. Goisman, P. L., "The Design and Analysis of a Complete Entity-Relationship Interface for the a Multi-Backend Database System," M. S. Thesis, Naval Postgraduate School, Monterey, California, December 1985.

12. Kloepping, G. R. and Mack, J.F., "The Design and Implementation of a Relational Interface for the Multi-Lingual Database System," M. S. Thesis, Naval Postgraduate School, Monterey, Califomia, June 1985.

13. Benson, T. P. and Wentz, G. L., "The Design and Implementation of a Hierarchical Interface for the Multi-Lingual Database System," M. S. Thesis, Naval Postgraduate School, Monterey, Califomia, June 1985.

14. Emdi, B., "The Implementation of a CODASYL-DML Interface for the Multi-Lingual Database System," M. S. Thesis, Naval Postgraduate School, Monterey, Califomia, December 1985.

15. Anthony, J. A. and Billings, A. J., "The Implementation of an Entity-Relationship Interface for the Multi-Lingual Database System," M. S. Thesis, Naval Postgraduate School, Monterey, Califomia, December 1985.

16. Hsiao, D. K. and Menon, M. J., "Design and Analysis of a Multi-Backend Database System for Performance Improvement, Functionality Expansion and Capacity Growth (Part I)," Technical Report, OSU-CISRC-TR-81-7, The Ohio State University, Columbus Ohio, July 1981.

17. Hsiao, D. K. and Menon, M. J., "Design and Analysis of a Multi-Backend Database System for Performance Improvement, Functionality Expansion and Capacity Growth (Part II)," Technical Report, OSU-CISRC-TR-81-8, The Ohio State University, Columbus Ohio, August 1981.

## INITIAL DISTRIBUTION LIST

| | | No. Copies |
|---|---|---|
| 1. | Defense Technical Information Center<br>Cameron Station<br>Alexandria, Virginia 22304-6145 | 2 |
| 2. | Library, Code 0142<br>Naval Postgraduate School<br>Monterey, California 93943-5002 | 2 |
| 3. | Chief of Naval Operations<br>Director, Informations Systems (OP-945)<br>Navy Department<br>Washington, D.C. 20350-2000 | 1 |
| 4. | Department Chairman, Code 52<br>Department of Computer Science<br>Naval Postgraduate School<br>Monterey, California 93943-5000 | 2 |
| 5. | Curriculum Officer, Code 37<br>Computer Technology<br>Naval Postgraduate School<br>Monterey, California 93943-5000 | 1 |
| 6. | Professor David K. Hsiao, Code 52Hq<br>Computer Science Department<br>Naval Postgraduate School<br>Monterery, California 93943-5000 | 2 |
| 7. | Professor Steven A. Demurjian<br>Computer Science & Engineering Department<br>The University of Connecticut<br>260 Glenbrook Road<br>Storrs, Connecticut 06268 | 2 |
| 8. | Deborah A. McGhee<br>232 Foxhunt Road<br>Columbia, S.C. 29223 | 2 |